THE UNIVERSITY OF ALBERTA

**RELEASE FORM**

NAME OF AUTHOR Darrell Douglas Makarenko

TITLE OF THESIS Simulating Computer Architectures Using the Architecture Description Language S*A

DEGREE FOR WHICH THESIS WAS PRESENTED Master of Science

YEAR THIS DEGREE GRANTED Fall 1982

THE UNIVERSITY OF ALBERTA

# Simulating Computer Architectures Using the Architecture Description Language S*A

by

Darrell Douglas Makarenko



A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE

OF Master of Science

Department of Computing Science

EDMONTON, ALBERTA

Fall 1982

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled "Simulating Computer Architectures Using the Architecture Description Language S*A" submitted by Darrell Douglas Makarenko in partial fulfilment of the requirements for the degree of Master of Science.

for my parents

## Abstract

Recent research in the area of computer architecture design has resulted in the awareness of a need to develop design methodologies and in fact to create a *design discipline* in this area. One such design methodology is based on the use of a *family of languages* for representing computer architectures at various levels of abstraction during the design process. The S* family of languages (DASG81a) consists, presently, of two related languages, S*A, for higher level architecture representations and S*, a language schema for microcode representations.

The main objective of this work has been to develop and implement a compiler and simulator for S*A. It provides a basis for some practical experience in using the family of languages approach for computer architecture design. As well, the exact nature of computer architectures, levels of abstraction, computer architecture description languages, simulation of computer architectures, and the design process itself are examined in detail to provide a further foundation towards future practical studies.

## Acknowledgments

I would like to express my appreciation to my supervisor, Dr. Subrata Dasgupta, for the guidance and feedback that he provided throughout all stages of this work.

As well I would like to express my thanks to all those who provided me with encouragement and especially to Lynne, for her continual love and support.

## Table of Contents

# List of Tables

# List of Figures

# Chapter 1

## Introduction

It can not be disputed that modern day computers are increasing in processing power at a rate that has never before been seen. Yet, in order to reap the benefits of this power, we have to deal with enormous increases in complexity, both in the circuitry and in the organization of computer systems. One of the places where complexity appears, and poses problems, is in the area of architecture design. In the past, many computer systems have been designed by relatively small teams of people and in an ad hoc fashion. As this is no longer a viable method, much work is currently being done (ZUR68,BARB80,CAVO81,DASG81a) to identify new design methodologies which will enable designers to handle the complex designs of present and future computers. However, though many good ideas have resulted from this work, it will take time for them to emerge from the research realm and move into the arena of "real" applications. Furthermore, while some encouraging practical results have been obtained, (BARB77a,BARB80,VAN81), many serious problems encountered in the architecture design process remain to be resolved. Hopefully, out of the work being done will arise a *design discipline* that can help to conquer the, as yet, unmastered complexity of the machines we are now building. This thesis is intended to be a step towards that goal.

In order to understand the contribution that this thesis attempts to make, it is necessary to first briefly look at the concept of using a *family of languages* for the design and implementation of machine architectures.

This concept results from the need to represent computer systems at various levels for the sake of human comprehensibility. These levels are often referred to as *levels of abstraction* because each level can be thought of as an abstraction of the entities in the level below it, the lowest level considered being the physically realized implementation or hardware. The basis of the family of languages approach is to provide not just one language for all design levels, but a group of languages, each one designed for a specific level. Furthermore, the languages are all related to one another in that certain similarities in the constructs exist among all the languages in the entire family. This facilitates the conversion of an architecture description in one of the languages at a particular level of design to a description of the same architecture, or its implementation, at another level using one of the other "kin" languages.

The family of languages known as the S* family has been devised as a basis for this approach (DASG81a). Presently, this family consists of two languages. The first is S* (DASG78), not a complete language in itself, but a language *schema* which provides a machine independent basis for describing the implementation of computer architectures at

the microcode level. The other language is S*A (DASG81b), and is designed for architecture descriptions at the higher architecture level including those involving problems of concurrency and synchronization.

The advantages of the family of languages technique manifest themselves best when it is incorporated into the design process itself. This involves the building of a design environment for the user so as to facilitate the use of these languages in the creation, testing, modification and verification of computer architecture descriptions.

Towards this end, the work that has already been done includes the following.

1. the implementation of the S* language schema with respect to the Nanodata QM-1 (NANO77) thus providing an *instantiated* version of S* (KLASS81a,KLASS81b), known as S*(QM-1). A compiler for the language S*(QM-1) has been partially written and implemented.
2. the development of the syntax and semantics of S*A (DASG81b), a language which includes constructs for handling problems of concurrency of execution and the contention for resources. This language is a kin language to S* (and thus to S*(QM-1)).

In continuation of this line of research, a major part of this thesis concerns the development and software implementation of a compiler and a simulator for S*A. Included with the simulator is the core of a design environment for the creation and testing of computer

architecture descriptions in the language S*A. It is anticipated that the application of this design environment may eventually lead to modifications to the S*A language itself. Any shortcomings that are not currently apparent will become so when S*A can be used in a more practical way.

Also included in this thesis are some discussions on the exact nature of the kinship that exists between S* and S*A. With compilers for both languages now implemented, as well as the simulator for S*A, the tools are available for the role of a *family of languages* as part of a computer architecture design methodology to be studied on a large scale using real examples. As is the case with other research in the realm of *experimental computer science*, only with this sort of experience can we fully develop this concept of a family of languages, to provide some real assistance in the design of future computer systems.

# Chapter 2

## Representing Computer Architectures

It is generally accepted that one of the most important aspects of any technique for designing computer architectures is the means by which the designer represents the description of an architecture both at the final stage as well as during its design. In response to this, numerous languages have been proposed with the objective of representing computer architectures (BARB77b,CHU65,DASG81b). In the case of already existing machines, the language is used to produce a fixed or static representation and might be used for formal verification, while in the case of machines that are not yet implemented, it produces an evolving or changing representation, being integrated into the design process itself.

How does a computer designer select the language which best suits the level at which he wishes to work? The first step is for him to define, in a clear way, the design level with which he is concerned. As design levels have traditionally been a somewhat vague notion and tend to lend themselves to intuitive definitions, it is necessary to examine what is actually meant by a *level* of design and the implications of that meaning. The exact nature of a level of design can be best expressed by breaking it into the following two components.

- the level of the basic entities of the design. We will refer to this as the *granularity* of the design level.

- The level of complexity of systems and components that the designer wishes to describe using the basic entities within realistic constraints. We will refer to this as the *field of view* of the design level.

The distinction between these two points is important so we will examine them in more detail.

1. Granularity of the design level

This describes the structure and behavior of the lowest level entities in the design level, i.e. the basic building blocks. The granularity of a design level may be classified into a number of areas.

- data entities - which may include files, variables, registers, bits, voltage levels, etc.
- operation entities - which may include such things as the logical AND operation, division operation, read file operation, subroutine call operation, stack operations, etc.
- timing and synchronization entities - which may include clock pulses, semaphores, priority handling of requests, parallel actions, sequential actions, etc.
- data manipulation entities - these may include concatenation, subscripting, truncation, padding with zeros, etc.

Basic entities may span a number of the above areas and may also include combinations of the above. For example, in some designs, the M68000 microprocessor chip

could be considered a basic entity of a design level. As well, basic entities can be of a more abstract nature. A list or a state, for instance, could be a basic entity. These entities belong to the design level and may or may not correspond to basic constructs in the computer description language used to represent the design.

2. Field of view of a design level

This idea best reveals itself as the apparent level of sophistication of designs that are described at the particular design level.

In the definition given above for field of view, the term *within realistic constraints* is important. A designer can, if he wishes to, use resistors and diodes to describe a data base management system. However, realistically resistors and diodes confine the designer to describing systems on the order of complexity of gates, registers, etc.

Remember, that the field of view refers to the level of the particular design that is being described, not to the level of the computer description language that it may be represented with.

In clearly defining the level he is working at, the designer must determine for all areas of his design what the *granularity* and *field of view* will be of his (possibly envisioned) design. The granularity that is described may correspond to a set of physical components such as resistors and diodes or it may correspond to abstract components like

files or processes. The maximum field of view of any design is in some sense limited by the realistic complexity constraints of the chosen basic entities. This does not however eliminate the need for the designer to define and be aware of the actual field of view he desires to work with.

It is important to have an understanding of these concepts as a means to clearly define a design level. Without such concepts, the designer often ends up with an intuitive feeling for his design level but will find difficulty conveying the exact nature of that level to others. In order to assist the reader in gaining a familiarity with these terms, a few examples will be given of traditional, reasonably familiar design levels in terms of granularity and field of view.

1. Circuit design level - The granularity of this level includes as basic entities, transistors, resistors, diodes, voltage levels and the electronic laws which describe their behavior. The design itself consists of the electrical interconnection of the above physical components. The field of view of this design level depends, of course, on the specific application but it is often restricted to a set of two or three variable logic gates or possibly minor combinations of them. In this case the field of view is much smaller than the maximum field of view possible, as one could design very complex circuits at the circuit level. This is seen more often in the case of analog circuits. We have chosen to

restrict ourselves to designing logic gates.

2. Logic gate design level - The granularity of this design level usually has as basic entities a restricted set (AND,OR,NAND,NOR,NOT) of logic gates along with associated gate delays. The design involves the electrical interconnection of these gates. The field of view is again application dependent but could include for example, registers, priority encoders, sequencers, etc. Again, the field of view is a constraint imposed by the designer rather than a real one as it is possible to design fairly complex machines using only nand gates. This does not imply though, that it should be done this way.

3. exo-architecture - This term is borrowed from (DASG81b) and means the level at which the assembly language programmer or compiler writer would view a machine. It is also refered to as the macro-architecture level. It is usually understood in terms of its granularity which is an array of memory cells, some data types, instruction formats, a set of instructions and some rules as to how they operate. The field of view of this level is usually constrained to smaller program modules but is really limited only by the ability of the assembly language programmer, which can be apparently unlimited if one examines some of the monstrous assembly language programs that exist. Quite often this level becomes the field of view of lower levels, and is a very

familiar level. This is because in the past this level has been the dividing line between hardware levels and software levels or where the manufacturer's responsibility left off and user's responsibility took over. Microprogramming is now more widely used, especially with user microprogrammable machines, software support provided by the manufacture is becoming more commonplace and a stronger link is developing between hardware and low level software of the operating system. Thus the exoarchitecture level is no longer as well defined as it once was.

Computer levels are many and varied and more often than not overlap. They are not always as clear cut as they may seem. As an example, if the designer were using open collector gates at the logic gate level, he may require resistors to describe his design and so resistors would have to be included in the granularity of the logic gate design level. One normally thinks of resistors as an entity found only in the circuit design level. Both an upper bound (field of view) and a lower bound (granularity) are needed to clearly describe a design level. This is necessary if one hopes to constrain a design to a particular level and convey the nature of that level to others.

Suppose now we have a design level for which we have defined both the granularity and the field of view and thus have a clear understanding of the level we are concerned with. It is assumed that the designer would like to

represent a design at this level in a formal way using one of the languages available for describing computer systems. How do we select which language is best to use? To answer this we will first examine more closely the process by which a language is used to represent a design.

The designer needs to provide a *mapping* between the basic entities of his design level and the basic constructs and operators of the computer description language that has been chosen. For example,

- a register may be mapped onto a variable
- a full adder may be mapped onto an add operator
- memory may be mapped onto an array
- sequential execution may be mapped onto statements
- parallel execution may be mapped onto a **PAR Begin** construct as found in Algol 68.

The designer needs some measure of quality of the language in representing the level of his design. One such measure would be the ease in producing this mapping. If the mapping is clumsy and unnatural then the designer should either try to produce a better mapping or else consider choosing a different language for the representation. This mapping can be of an informal intuitive nature or can be formally described. It is beneficial to study this idea of a mapping in some detail as it will not only aid in the choosing of a language to represent a design level but it will also give us some insight into the type of constructs that should be in a language to facilitate a good mapping.

What does a *good mapping* consist of?

1. It should be simple. There should be many cases where the basic entities of the design level map directly onto basic constructs of the language.
2. It should be complete. All basic entities in the granularity of the design level should be included in the mapping. The mapping should allow the designer to describe everything in the field of view of the design level. Note however that it is not necessary to use all of the facilities of the language.
3. It should be natural. It should be possible to formally describe the mapping, though it is not always necessary to do so. It should not go against a designers intuitive thinking. This is especially important when the mapping is not formalized and must be remembered by the designer.

No computer description language, for example, has registers in it. It may contain constructs onto which a designer can easily map registers, but these constructs only represent the registers which may or may not physically exist. It is the mapping process that defines what constructs will be used to represent the possibly envisioned physical entities. In light of an awareness of this mapping process, let us examine the use of conventional high level programming languages such as Algol or PL/1 for describing computer systems.

Conventional programming languages may be used as computer description languages for a number of reasons. Their constructs are usually of such a general nature that for many design levels it is easy to map the basic entities onto these constructs. There are, of course, the advantages of availability, reliability, portability, familiarity etc. There are many cases however where conventional programming languages are unsatisfactory. This is because it may be impossible to map the basic entities of certain design levels onto the constructs which are available in conventional programming languages. More often though, the mapping is possible but it turns out to be an awkward unnatural mapping. As an example, it is possible to represent the logic gate design level in Fortran with each logic gate being mapped onto a logical subroutine with two logical inputs. However one can easily see that representing computer systems at the logic gate design level in Fortran with this mapping would be very clumsy.

The major problem with available conventional languages though is the distinct lack of constructs to easily represent basic timing entities that are an integral part of computer descriptions at all levels. Most conventional languages are awkward to use when representing clock pulses, parallel execution of events, synchronization, etc. It is primarily to overcome this problem that numerous special purpose computer description languages have been invented with constructs onto which timing and other entities

associated with computer design levels can easily be mapped.

In the S* language schema, the mapping process has been formalized and is referred to as instantiation (KLASS81b). The design level that the designer wishes to use is the micro-architecture level of some microprogrammable machine such as the QM-1 (NANO77). The functional units of the QM-1, at its micro-architecture level are mapped onto declarations of variables and operators that are user written in the S* language schema. The original S* language schema, along with S* descriptions of QM-1 components, are then referred to as the language S*(QM-1)

This process is really the same as, for example, the representation of memory in an ISPS language description as the declaration MEM(0-4095) (BARB77b). The only difference is that in S* the designer is forced to formally describe the mapping of the design level's basic entities onto constructs in the language schema S*. This specific process which is done before the writing of S* computer descriptions is even started, is known as instantiating the language onto some microprogrammable machine. In ISPS and most other computer description languages the mapping is often done informally and as the need arises in the design process.

In summary, it can be seen that it is important to understand what is involved in using a language to represent a computer architecture. The designer must be aware of the *granularity* and *field of view* of the design level he is concerned with as well as the relationship between the

design level and the computer description language. In many cases the language description of the architecture will become the defining representation of the architecture and will be used for simulating, testing and verification of the architecture. In later chapters we will look at how the computer description language and the mapping of the design onto it can become an integral part of the design process itself.

# Chapter 3

## Comparison of a number of CDDLs

This chapter provides an overview of some of the work that has been done in the area of computer design and description languages (CDDLs). We will present a fairly detailed description and comparison of a few of the languages that have been developed along with a general discussion on some of the others. We will include in this discussion a look at how these specific languages and their constructs relate to the idea of a mapping between the language and the design level.

There are almost as many computer description and design languages in existence as there are researchers but we will concentrate on those that have, for various reasons, become well publicized in the research community. The popularity of a computer description language, as is often the case with conventional programming languages, may depend upon such things as the availability of compilers, simulators, etc., rather than on the merits of the language alone. Thus no claims are made as to the merit of the languages we have chosen over the ones we have omitted, the choice being for comparative purposes only. It is very difficult to produce a quantitative ranking of computer design and description languages as each language will contain traits and features specific to the distinct level it may have been designed for. However, we shall see that each level of abstraction in computer design requires

certain features regardless of the language involved.

The emphasis of this study will be on those languages that have been used to represent designs at the register transfer level (RTL) (BARB75). This is a level of computer architecture that falls somewhere between the logic gate level and the level seen by the assembly language programmer. Typically it is characterized by discrete data items (usually bits) being transferred between storage entities in discrete time intervals. The vagueness of the exact nature of the level is attributed to the need to divide it up, especially in the novel and complex computer designs of today, into a number of different levels. The design decisions that now occur between the logic gate level and assembler programmers level are very complex and require more than one level to represent them. When we refer to higher level designs and architectures and lower level designs and architectures we will be speaking in a relative sense within the constraints of the broadly defined register transfer level. This will should also allow us to come to some conclusions as to which constructs are best for representing certain entities and also those entities which are not easily represented in any of the languages.

For this detailed study we have chosen the languages S* (DASG78) and S*A (DASG81b), ISPS (BARB77b), SLIDE (PARK81), and DDL (DULEY68,BREUR75). We will see that all of these languages are basically procedural languages with many constructs similar to those found in conventional

programming languages. Nonetheless, each has its own special features which the inventor deemed were unique and necessary enough to warrant the creation of *yet another* computer description language. The following is a brief description of each language.

### 3.1 S* and S*A

These two languages were chosen to be a part of this study primarily to provide a basis for future discussions on the relationship between *kin* languages in a family of languages (DASG81a). Furthermore, the implementation of a software simulator for S*A forms a large part of this thesis.

S* is a language specifically designed for writing microprograms for microprogrammable computers (DASG80,KLASS81b). The main goal of the language is to provide a syntax that could be used to describe microprograms for a variety of machines. S* is really not a complete language in itself but is a language *schema*. The entities that are fully defined in S* include basically a set of primitive and structured data types. These provide a basis for declaring microprogrammable data objects. There are also a number of statement forms including case, branching, looping, etc. The exact semantics of these statements are only partially defined in the language schema and are dependent on the microarchitecture of the machine

that the language will be implemented on. There are also constructs for expressing parallelism in the execution of statements. The reasoning behind the language schema approach is to overcome the obstacle of the large differences and peculiarities of and among different microprogrammable machines. The idea of leaving some of the semantics of statements only partially defined, until the instantiation of the language on a specific architecture, was considered to be the best way to achieve the goal of a machine independent microprogramming language (DASG80).

S*A is a kin language to S* and thus has very similar constructs but is, in contrast, a completely defined language. It is basically structered for representing computer architectures at a level known as the *endo-architecture level* which lies between the register transfer level and the assembler language programmers level of design. It can also be used however, to provide architecture descriptions at the assembler language programmers level or *exo-architecture level*. It is quite a powerful language with a variety of constructs for looping, branching, cases, procedure calls, etc. Its strength lies in its ability to represent architectures involving parallelism and resource contention as might be found in some of the novel designs of interconnecting processors. The language also provides a block structure to facilitate segmentation and understandabity, to limit scopes of variables and to allow the creation of critical sections in the code.

Both of the above languages can be used to represent the structure as well as the behavior of the architecture that they are describing. The S*A simulator provides a design environment for the simulated execution of architectures described in that language. This will be discussed more in a later chapter.

### 3.2 ISPS

The ISPS language (Instrucion Set Processor System) was chosen as one of the better developed and publisized register transfer level languages. It is one that has been used in a number of real applications (BARB80,VAN81), and has been well publicized. It can describe both the structure and the behavior of a machine. It includes most of the constructs and facilities available in conventional high level programming languages. These include declaration statements, arrays, text strings, variety of representations for data (e.g. 2's complement), etc. In the area of program control the language includes IF structures, looping(repeat), go to's (leave), and a case structure (decode). All of the usual arithmetic and logical operations are present, as well as concatenation, shifting, assignment, etc. The language is well structured and subroutines can be written to both accept parameters and return values. Statements can be defined to either execute sequentially or in parallel.

The language was designed to be simple and flexible and to this end it succeeds. Its simplicity however, can make it somewhat burdensome for complex systems. In addition, there are a few constructs (e.g. semaphores) necessary for modern architectures that are noticeably missing. Some of these facilities are available in the simulator developed for the language.

The compiler for ISPS produces a global data base tree (GDB-tree) which is a compact, symbolic, reversible, tree structure representation of the ISPS description. The compiler checks for syntactic correctness only. The GDB-tree can then be used for various applications one of which is simulation. An ISPS description may also contain user-invented attributes which the compiler basically just ignores and passes to the GDB-tree.

The ISPS simulator (BARB78) allows one to take the GDB-tree and *execute* it. It converts the GDB-tree to register transfer machine (RTM) code (for an imaginary register transfer level computer) and executes the RTM-code on a software simulated machine. It also allows the user to examine and redefine ISPS variables during the simulation, set breakpoints, trace the values of variables, supply test data, count variable usage, simulate parallelism and critical sections, associate execution times for procedures and operations, etc.

### 3.3 SLIDE

SLIDE (PARK81) is a language specifically designed to describe I/O type operations and interconnections at the register transfer level. It is very similar to ISPS but is basically designed for interconnection behavior descriptions. All features of ISPS are in SLIDE except the case statement, procedure parameters and a few logical operations.

The fundamental interconnection behavior which SLIDE as an I/O description language, tries to model includes:

- concurrent execution of multiple processes;
- contention for shared resources between processes;
- critical sections of execution within processes;
- simultaneity of execution both within and across processes;
- processes which behave in a subordinate fashion with respect to other processes and
- interconnections which exhibit behavior.

A SLIDE program is structured into processes and subprocesses all with global and local variables. Processes will begin execution non-procedurally, that is when certain conditions become true. As well there is a priority mechanism among processes at the same level. Only one process at a given level may be executing at a given time.

Some of the major features available in SLIDE that are useful in I/O descriptions are the Delay statement, a synchronization facility, the Iferror statement, parallel

execution of statements and accessibility to bit slices of data. Various other available features useful in describing systems are FIFO buffers, combinatorial logic, associative memory tables, high-low and low-high transition descriptions and format operations.

Parameterization of descriptions which are bound at simulation time are possible. As in ISPS the SLIDE compiler also produces a GDB tree which can be used for simulation, program verification and various other applications. A SLIDE simulator has been written (ALT79) and will be described in detail in a later chapter.

### 3.4 DDL

This language (DULEY68,BREUR75) was chosen as a representative of RTL computer description languages that are very close to the hardware level and is consequently a very hardware oriented language. Many of its constructs are specifically designed to represent, and in fact are given the same names as, real hardware components. The constructs in the language include such entities as latches, registers, memory, terminals, etc. The usefulness of this type of description, so tightly bound to the hardware components, is somewhat questionable with the technology of the hardware in computers changing so quickly these days.

The level of design that one can produce using the language is limited to fairly simple or modular circuits. If

one attempts to design complex circuits that are not of a regular nature, the description can become too complex to work with since the primary data item is the voltage level. However, as a means of describing symbolically, what it had previously been necessary to describe pictorially, the language works well.

### 3.5 Others

There are a number of other hardware description languages existing which will be just touched on briefly here.

- ADL (LEUNG79) is an architecture description language that was specifically created for the design of architectures based on the concept of data flow computers.
- APL (APL74) is a powerful general purpose programming language with many constructs that have facilitated its use as a description language for architectures. It is however, of a very mathematical nature and can only be used for highly functional descriptions such as state transition descriptions.
- CDL (CHU65) was one of the earlier design description languages that became reasonably successful for simulation and design automation.

There are of course others that we have not mentioned.

### 3.6 Comparative study of CDLs

While it is impossible to create a language which can be used easily to represent all levels of a computer design, most languages do manage to be acceptable for representing more than one level. This may be the result of a specific goal of the designer, or may just result from the ubiquity and variety of modern day computer design levels. Using a single language to represent two distinct levels of design, particularly adjacent levels, may or may not be the best approach as we shall see later in our discussion of the notion of a family of languages. Regardless of this, the languages we have chosen do overlap to a large degree in the levels that they represent and thus comprise a good set for comparative purposes.

This comparative study consists of an item by item comparison of the five languages that were chosen, including a summary of their relative merits, disadvantages, similarities, differences and goals. This format was chosen to emphasize the similarities and differences of the languages rather than teach the reader how to create designs in any one language. The items specifically covered are goals of the languages, expressions and operators, assignment statements, primitive data types, structured data types, block structure, control statements, synchronization and timing constructs and primitives, detection of transitions and procedural calls.

### 3.6.1 Goals

1. S* is a language schema and cannot really be considered a complete language until it has been instantiated with respect to a particular machine. This instantiation process consists of mapping the entities of a particular microprogrammable machine at the microarchitecture level onto the constructs in the schema. The major goal of the language was to provide a language schema which can represent the microarchitecture level of different microprogrammable machines. Due to the diverse nature of machines at the microarchitecture level, it was necessary to use a schema rather than a full language.
2. S*A is an architectural description language. It has been designed to represent computer systems at a level slightly above what has been known as the register transfer level. It includes constructs which can handle the synchronization problems of configuring multiple CPUs and other forms of parallelism found in modern day computer systems.

   Both S* and S*A were designed as kin languages and make up a family of languages known as the S* family. The constructs in both languages are oriented towards formal verification of algorithms and designs described in the languages, and proof rules for all of the constructs have been developed.
3. ISPS is oriented towards the description of a large

class of designs at the register transfer level designs. The language was meant to be appropriate for diverse applications such as design, simulation, verification, etc. Flexibility and simplicity were its primary goals which it has achieved reasonably well.

4. SLIDE is a hardware description language designed for the representation of input/output interfaces and interconnected digital systems. Its specific goals are the description of asynchronous, concurrent processes which can communicate with one another. It is basically a RTL language and can also be used to represent simple sequential systems at that level.
5. DDL is a register transfer level language that is very close to the logic gate level. Its constructs are often closely related, if not equivalent in some cases, to actual hardware components. DDL allows for a fairly easy transformation from its constructs to a design involving logic gates. It is basically a convenient language for representing what had previously been represented using logic diagrams.

### 3.6.2 Expressions and Operators

Table I, which follows, illustrates the various operators that are available in the different languages. All the languages have addition, subtraction, shift operators, relational operators, etc. which are necessary at almost all levels of design. Notice that the special

Table I - Operators

| Operations | S* | S*A | ISPS | SLIDE | DDL |
|---|---|---|---|---|---|
| Relational | yes | yes | yes | yes | yes |
| Arithmetic | yes | yes | yes | yes | yes |
| Shift | yes | yes | yes | yes | yes |
| Bitwise logical | - | yes | yes | yes | yes |
| Modulus | - | - | yes | yes | - |
| Concatenation | - | - | yes | - | yes |
| Parity | - | - | - | yes | - |
| Increment | - | - | - | - | yes |
| Decrement | - | - | - | - | yes |
| Selection of bits | - | - | - | - | yes |
| Reduction of bits | - | - | - | - | yes |
| Extension of bits | - | - | - | - | yes |
| Push and Pop | yes | yes | - | - | - |
| Assoc. array | - | yes | - | - | - |

operators in DDL are for bit manipulations, something one would have need of at such a low level. S*A on the other hand does not have such bit oriented operators, (i.e. concatenation is noticeably missing) but has instead much higher level operators such as push and pop stack operators. S* schema leaves operators only partially defined as they are dependent on the machine it will be instantiated on. SLIDE deals with I/O communications and thus includes such things as a parity operator which is very specific to this

application. In summary we see that the type of operator that a language contains as well as the types of operators that it does not contain can tell us to some extent at which design levels the language is meant to be used. By excluding certain operators, the inventor of each language tries to confine the use of the language to the levels for which it was designed.

### 3.6.3 Assignment Statements

Assignment statements are present in all of the languages. Since we are dealing with languages that represent design levels around the register transfer level, it is not surprising to find that they all contain a specific construct to handle the transferring of data. ISPS, though, is a little more powerful in this area, allowing for such things as concatenation on the left hand side of the assignment statement and multiple transfers per statement.

### 3.6.4 Primitive Data Types

The bit is the most primitive data type in all of the languages, and in S* and S*A it is explicitly stated as such. DDL however has numerous other primitive data types as well, including terminal, register, memory, latches, time, delay, boolean and element. In DDL we see that the bit is represented in various forms depending whether it is part of

a terminal line, register, latch etc. In the higher level languages this distinction is not made and the physical implementation of a bit is ignored. It is considered to be simply a bit of information stored in some unspecified manner.

### 3.6.5 Structured Data Types

Table II, which follows, gives a brief overview of the structured data types found in each of the languages.

Table II - Structured Data Types

| Data type | S* | S*A | ISPS | SLIDE | DDL |
|---|---|---|---|---|---|
| bits | yes | yes | yes | yes | yes |
| arrays | yes | yes | yes | yes | yes |
| tuples | yes | yes | - | - | - |
| stacks | yes | yes | - | - | - |
| assoc. arrays | yes | yes | - | yes | - |
| registers | - | - | - | yes | yes |

All the languages use sequences of bits and arrays as they are the standard forms for storing computer data. DDL also has the ELEMENT declaration which can declare an item whose structure and behavior are left unspecified except for its I/O connections. The noticeable difference among the languages is in the occurrence of higher level constructs such as tuples (equivalent to records), stacks and

associative arrays in the S*A language. In S*A one is more likely to use these higher level constructs to describe the data that is used so as to hide the details of its storage from the design level.

### 3.6.6 Block Structure

Due to the hierarchical nature of computer systems all the languages have found it necessary to have constructs to support block structures. Each language may use a number of different types of constructs to implement varying kinds of block structures.

- S* uses programs and procedures;
- S*A uses systems, mechanisms procedures and functions;
- ISPS uses the Begin -- END construct;
- SLIDE uses the Begin -- End construct as well as processes;
- and DDL uses the SYstem, AUtomaton and SEgment constructs.

### 3.6.7 Control Statements

There are a variety of control statements in the languages as shown in Table III.

Note that the different languages may not use the same name for functionally identical constructs. ISPS, for example, uses a "decode" statement for what is usually

Table III - Control Statements

| Stmt. type | S* | S*A | ISPS | SLIDE | DDL |
|---|---|---|---|---|---|
| If | yes | yes | yes | yes | yes |
| While | yes | yes | - | yes | - |
| Repeat | yes | yes | yes | yes | - |
| Case | yes | yes | yes | - | - |
| Call | yes | yes | yes | yes | yes |
| Return | yes | yes | yes | yes | yes |
| Go to | yes | - | yes | yes | yes |
| Act | - | yes | - | - | - |
| Exit | - | yes | - | - | - |

called a "case" statement. The reader should note that DDL contains no direct constructs for the specification of loops. Loops however can be constructed from the other constructs using states. In the higher level languages looping is a must for computer designs of any complexity. The Act statement found in S*A is a type of procedure Call statement where the calling routine does not wait for the return of the procedure before continuing execution (see Appendix C). The Exit statement terminates execution of an Activated procedure.

### 3.6.8 Synchronization and Timing Primitives and Constucts

- In S* there are the **sig** and **await** primitives for synchronizing between processes. As well, the **cocycle**, **stcycle** and **region** constructs are used for synchronizing statements that are executing in parallel within a process.
- In S*A, for inter-process parallelism and synchronization, either the mechanism construct can be used or else the lower level **sig** and **await** semaphores can be used. Parallel execution of statements within a process is also allowed.
- ISPS allows for both sequential and parallel execution of statements within a process but contains no constructs for inter-process synchronization.
- SLIDE offers the delay and sync primitives to synchronize between processes as well as allowing parallel execution of statements within a process.
- DDL does not have direct facilities for parallelism or synchronization of any kind. It does have STate descriptions however, allowing parallel events to occur in a very limited sense.

Our major concern, in this thesis, is the ability of a language to handle parallelism. As SLIDE is designed for representing I/O operations, which typically employ a lot of parallelism and synchronization, it is well equipped with low level synchronization constructs. Parallelism between

processes is quite important at the higher architectural levels which is why S*A contains constructs specifically designed to handle the problems of mutual exclusion etc. The absence of constructs of this nature in ISPS, S* and DDL was probably a design decision rather than an oversight as the concept of inter-process communication is a high level one and would not fit in well in a lower level design language.

### 3.6.9 Detecting Transitions i.e High - Low

This is a useful concept that is only seen in two of the chosen languages. S*A employs it, via the channel construct, at the higher architectural level. Slide also has this facility, which is not surprising considering the specialized I/O nature of the language.

### 3.6.10 Procedure Calls

Table IV summarizes the facilities in the various languages with respect to procedure calls.

Recursive calling of procedures is not allowed in any of the languages. This is not surprising as the languages are designed to represent hardware, or abstractions of it, which is not recursive in nature. Slide is the only language that allows for non procedural activation of procedures. One of the key decisions seems to be whether to allow parameters or not. It is really a design decision as to how complex the

Table IV - Procedure Calls

| Facility | S* | S*A | ISPS | SLIDE | DDL |
|---|---|---|---|---|---|
| Procedure calls | yes | yes | yes | yes | yes |
| Input parms | - | yes | yes | - | yes |
| Output parms | - | yes | yes | - | - |
| Return value | - | yes | yes | - | yes |

inventor of the language wants the language to be. Note that by using global variables you can always get around the need for parameters but at a cost of added complexity to your design.

### 3.7 Summary

There are many other items which can be compared among the languages but these are the basic ones. Features such as synonyms, different number bases, etc., are notational conveniences in a language. They are useful and add flexability and thus should really be present in languages of all levels. Their absence is usually the result of time constraints rather than a design decision. There are also other features such as priority of processes and interruption of processes which are details of the scheduling of processes during the implementation of the language or simulation of the language and these areas will be discussed later. Among the different languages and the different design levels that they represent the major

difference that we are concerned with is in the area of timing and synchronization.

The lower level languages deal with clock pulses and statement parallelism while constructs in the higher level language of S*A deal directly with the problems of inter-process parallelism. S*A has tried to isolate the user from low level timing concerns which account for a lot of the complexity of lower level designs. S* is lacking in a lot of constructs as it is a schema and necessary constructs are built using the available ones during the instantiation process. Prior to the creation of the S*A language, none of the languages available could satisfactorily represent the higher level data structures of a high level architecture design while at the same time handle complex timing and synchronization considerations. With the use of parallelism being so prevalent today and many computer systems using multiple CPUs these constructs are very necessary at the high level architecture design level and can no longer be restricted for use by the I/O interface designer.

In summary, all of the languages are reasonably general purpose in their nature which provides flexibility of their use. As well they all use a lot of constructs that are familiar to us from conventional programming languages. However, very little work has been done on how these constructs should fit into the design process.

# Chapter 4

# The Design Process

In this chapter we will delve deeper into the exact nature of the process involved in the design of computer architectures. The development of computer design and description languages was a step towards simplifying the process of designing computers but it is not the total solution. It is also necessary to develop accompanying design methodologies in order to utilize these languages to their fullest.

## 4.1 Computer Architecture Design

The state of the art in computer architecture design is at a point now where a leap ahead is very much needed. The technology for computer hardware has advanced at an explosive rate. Chips have very quickly moved from MSI to LSI and now to VLSI. These advances however, have introduced the problem of designing computers and computer systems of such a complexity that merely understanding the design becomes difficult. Multiple CPU systems involving large amounts of parallelism are now commonplace. It is almost needless to point out that debugging, optimizing, and verifying the correctness of such designs is a major problem that will be difficult to overcome.

Let us examine the process that occurs during the design of a computer architecture. It can be broken down

into a number of basic parts.

1. the original specification of what the machine is to do when it is completed

   This specification may include such information as the speed, size, cost, chip count, instruction set, etc. Most designers realize, from the experience that has been gained in the field of software engineering, the importance of fully specifying the design before commencing to build it (PARN72).

2. a general conception of the high level layout of the machine along with a number of high level decisions

   These decisions may answer such questions as whether the computer will be built using microprogramming, random or bit-slice logic, whether instructions will be pipelined or not, the amount of parallelism that will occur within the machine, and others. It is possible to delay some of these decisions to a later stage.

3. the production of a detailed functional description of the machine to meet all of the required specifications decided about in step #1

   The description should be of a sufficiently low level and should include all timing constraints to facilitate the implementation of the machine based on the description.

4. the testing and possible verification of the design

   This will involve iterations of the above steps to produce a final version of the design. Not only is

logical correctness important here but the designer may also wish to optimize the design on some criteria in order to meet the specifications that were originally given.

5. the implementation of the design using one of the hardware technologies currently available
   This process should not be too difficult (possibly automated) as the design of the machine should have been thoroughly tested in the preceding step using simulation or some such means. It is conceivable that certain hardware technology based problems may arise and be of a serious enough nature that it becomes necessary to return to the previous iterative step to modify the design.
6. final testing of the complete machine

By far the most complex of these steps are 3 and 4 as they may involve a number of iterations to achieve the desired result. It is this area that we will concetrate on in this chapter. Specifically, we are dealing with the designs of computer architectures and an important basis is to have some method of representing the architecture in a human comprehendible format. This representation must be suitable for all aspects of the iteration procedure including testing, simulation, verification, etc.

Recent research trends have been, and understandably so, aimed towards the use of specially designed symbolic languages to represent the computer architecture. There are

many advantages to the use of languages rather than pictorial representations. Since the designs tend to be very large quite often a computer is used to store and manipulate the representation during the design process. Languages tend to lend themselves well to this sort of manipulation as computers are presently designed for symbolic input, output and manipulations. Furthermore the simulation of language based architecture descriptions is not a task of unsurmountable difficulty, given the state of the art in that field. Though pictorial representations can be more descriptive and easier to understand, they are often not as precise. With the familiarity that present day computer scientists have with a variety of programming languages, the concepts used in a computer architecture description language are very easily grasped. One final point is that languages tend to lend themselves well to formal descriptions of constructs which can become more important as the field of architecture verification begins to advance.

There is often some uncertainty as to what is exactly meant by the *architecture* of a computer. The term is defined in the The Funk and Wagnalls Standard College Dictionary as the "arrangement or ordering of a system's internal components". However, in addition to the aforementioned meaning, the architecture of a computer also includes a description of the nature of the hardware and/or firmware processes active in the computer as it performs its task. These processes are, of course, implicitly defined by the

arrangement of the components but are not necessarily obvious. Typically people will view a computer architecture from the assembly language programmers level or *exo-architecture* level, ignoring internal structure.

When using a language to describe an architecture, it is necessary to ascertain exactly what it is that is being described. Is it the components of the architecture and their interconnections or is it processes or functions that they perform without mention of their physical arrangement? This is important as it may dictate the type of language that will be used. The terms *functional*, *operational* and *structural* can be used to classify computer description languages along this line. These terms are not necessarily distinct and may overlap.

1. A language is said to provide a *functional* description of an architecture if it describes the results that the architecture produces without any indications as to how the architecture produced the results or the processes are involved in their production. The functional description is often of an abstract, axiomatic or mathematical nature. This type of description, though not the best for constructing a computer, is very useful in the area of architecture verification where proof rules may accompany the constructs of the language. Towards this goal proof rules for all of the constructs in S* and S*A languages have been developed.
2. An *operational* description of an architecture varies a

little with the preceding type. In an operational description there is an indication as to *how* the result is obtained. It is a "what happens" type of description, listing the basic operations that make up the high level description. It will convey a clear understanding of the steps involved in an operation or process. Often it is very simple to take an operational description of a machine and produce a simulation of it.

3. The next type of description is the one that has been in existence the longest, that is the *structural* description. This type of description will involve the components of the architecture and their interconnections with each other. It is this type of description that plays the major role in converting the design representation to its actual hardware realization. However, from the designers point of view it can be difficult to understand this type of description in terms of what the architecture is doing.

One aspect of the nature of computer architectures which is represented by the above categories is the *behavior* of an architecture. The behavioral nature is more dynamic in the sense that it will involve questions concerning how the architecture will perform in operation. Questions such as "How does information filter through the network?", "How are instructions decomposed to improve parallelism?", "How is pipelining used to increase execution speed?", "How are memory requests queued for sequential processing?", etc.,

are of a behavioral nature. The answers to these questions may involve either functional or operational descriptions but from a behavioral viewpoint. This is an important type of description as computer architectures are not static entities and their dynamic properties, while running, are of prime consideration in specifying them.

Thus there are a number of terms which can be used to describe the nature of a representation of a computer architecture. It is readily obvious that it is necessary to utilize ideas from all of these terms when producing a design. Some computer description languages though, tend to contain constructs and syntax which emphasize one or another of these categories.

DDL, as an example, is a hardware oriented language and represents logic gates and their interconnections. Its descriptions are very structural in nature thus making it difficult to use for large system descriptions. It can also be used to produce low level behavioral descriptions of combinations of logic gates.

S*A, on the other hand, provides for higher level computer architecture descriptions and thus is less structural and more functional in nature. As descriptions in S*A are algorithmic and are meant to be simulated, they also describe the behavior of the architecture. The statements in S*A do not necessarily represent the primitive operations that will occur in the architecture when an operation or process executes. The user is insulated from such structural

and operational entities as logic gates, clock pulses, subscript calculations, memory accesses, etc. The user may design an architecture in S*A without deciding on the hardware technology (i.e bit-slice, random logic, microprogramming) that will be used to implement it. The language is oriented towards a higher level of design with emphasis on such concepts as verification of the architecture, thus producing a more functional language.

The incentive behind studying these classification terms of design languages is twofold. First, it gives the reader a better comprehension of this somewhat vague area of computer architecture design. It can be seen now that not only are there different *levels* of designs (as was discussed earlier in this thesis) but at each level there are different *types* of designs (e.g. low level structural design, low level functional design, etc.).

The second reason for studying these classification terms is to provide some background for a discussion on the *family of languages* approach to architecture design which follows in the next section. As we begin to study the transformation of a design from one language representation to another, it will become necessary to be aware of exactly what elements of the design are present in each language representation and what elements or details are really injected during the transformation process itself.

## 4.2 The Family of Languages Approach

In the preceding section we noted that our primary interest in design process is in the stage involving iterations of designs going from a high level general specification of the architecture to a lower level. This lower level is a detailed final version which is logically correct and acceptable with respect to the original specifications. This is often considered one of the most difficult parts of the design process and is an area where tools to aid the designer should be graciously welcomed.

One such tool that has been developed is the idea of a design environment based upon a *family of languages*. Usually, the design itself is of such complexity that it is not possible to produce directly, a description of the architecture of sufficient detail to allow its realization in hardware. It becomes necessary then to subdivide this design into a number of different levels. Each level would be defined in a manner utilizing the concepts of *granularity* and *field of view* that were introduced in chapter 2.

The idea underlying the family of languages approach is that there would be a different language for describing the design at each of the specified levels. The languages would all have specific constructs and structure necessary for the type and level of the description that they would be used for, but there would be many similarities in syntax and structure throughout the languages.

These similarities produce a sort of *kinship* between member languages which serves a number of purposes. First, it allows one to comprehend all of the levels of a design as they will all have similar syntax based descriptions. Secondly, and more importantly, these similarities, especially between languages designed for adjacent levels of design, will allow the transformation of the design description from the higher level language to the lower level one to be intellectually manageable. In some cases, portions of this transformation process could be automated, as we shall see shortly.

Let us have a look at a description of a typical design process that one might use, utilizing the family of languages approach.

1. Define a set of specifications for the architecture based on some set of criteria as a measure of quality of the architecture design.
2. Choose a high level in which to represent the architecture and define (at least informally) the *granularity* and *field of view* of this design level in the sense that they were discussed in chapter 2. The level that is chosen should be one that can be easily represented by one of the high level languages in the family of languages.
3. Choose one of the high level languages from the family and map the basic entities of the design level onto the constructs in the language. For example, a data bus

might be mapped onto a global variable, a bus arbiter might be mapped onto a procedure, etc. The designer should also choose the lower level languages that he wishes to use based on the conceived method of implementation and examine the constraints that will result when mapping constructs from the higher level language onto the lower level one.

4. Using the higher level language, produce a high level description of the system. This design will often be of a more behavioral and structural nature rather than operational. The designer must restrict himself to the use of constructs in the language to which map into constructs in the lower level language.
5. Using the similarities of the languages within the family and the relationships that exist between the constructs in languages of adjacent levels, convert the high level language description to a description of considerably more detail in the next lower level language. Hopefully this process will be a relatively smooth mapping but it will involve, of course, a number of ad hoc conversions and optimizations.
6. Repeat step 4 to successively lower levels until a hardware or microcode implementation level is arrived at.

The above steps would of course all be iterated a number of times as part of the general design process described in the first part of this chapter. You will note

that the usefulness of this design process relies heavily on the ease with which one can represent the design at the various levels as well as the ease of converting designs of one level to designs of a lower level in a *kin* language. Even with similar languages this may not always be a trivial process. Remember, the languages will not be too similar as they must represent different types and levels of design. The ease of the conversion process between levels is a measure of quality for a given family of languages.

In the S* family of languages, there are presently only two languages. S*A is the higher level architecture description language and S* is a language schema for the writing of microprograms. Currently, S* has been instantiated onto the Nanodata QM-1 (NANO77), a microprogrammable computer, thus producing the language S*(QM-1) specifically tailored for writing microprograms for the QM-1. The S* family of languages will hopefully include, in the future, other languages. These languages could fall between the existing languages in terms of their design levels or perhaps there could be an entirely new branch in the hierarchical structure of the relationships between the languages in the S* family. As it is now, the family is designed to produce a final microcode representation of the architecture to be run on the QM-1. With the addition of new languages, S*A could also be mapped onto a lower level language specifically designed for, say, bit slice implementation. Another possibility is for the addition of a

language at a level higher than the present S*A. This would be for the representation of architectures at a gross functional or structural level without any details of the operations of individual components. These are areas for future work, outside the scope of this thesis and we will concentrate only on the existing two languages.

Let us return to the process of converting an architecture description in the language S*A to one in the language S* (or in fact S*(QM-1), the instantiated version). In order to simplify this conversion, what must be done is to develop a mapping from the constructs in S*A to the more detailed constructs in S*(QM-1). This of course will not be possible for all of the constructs as S*A can be quite sophisticated at times and S*(QM-1) is of a lower level nature.

There are some cases though where this mapping is very straightforward. For example, in the declarations of variables there are many cases where there are similar types of variables in both languages and the mapping between them is direct (e.g. arrays in S*A may map directly onto arrays in S*(QM-1) ). This direct mapping occurs in some of the statement constructs as well. The sequential statement delimiter in S*A is represented by the same symbol in S*(QM-1). The same holds true for the parallel statement delimiter. The assignment statement is, again, another example where there is a direct mapping between the two languages.

Unfortunately, all of the constructs do not map quite as easily as the examples given so far. The *if* statement, for instance, requires a little more manipulation. In S*A the *if* statement is of a general form, including an arbitrary number of *elif* clauses. The *if* statement in S*(QM-1) consists of a simple if (cond) . . . fi format. It should be clear to the reader that converting a complex *if* statement in the general form to a number of single statements in the simple form is not a complex task. A similar thing can be done with the *case* statement. The *case* statement in S*A can easily be mapped onto a series of single *if* statements in S*(QM-1).

All of the constructs examined so far have had a fairly simple mapping between the languages. Now we will examine some of the ones where the mapping is difficult or can even be impossible to obtain. The *call* and *act* statements are available in both S*A and S*(QM-1) and so it would appear that there might be a direct mapping. This is not the case, however, for a number of reasons. In S*(QM-1) only one level of subroutine calls is allowed while in S*A, using private procedures, multiple levels are allowed. This problem could be overcome by various means. The simplest, though not the most efficient, would be to use complete symbolic substitution in S*(QM-1) to implement private procedure calls in S*A. This would not be totally unreasonable if the design was for simulation only rather than implementation as there is no recursion allowed in subroutine calls. This

solution though, is far from elegant.

A more substantial problem occurs with the mechanism of the *call* or *act* statement in S*A when parallel paths of execution are involved. The semantics of the S*A *call* statement dictates that sequential execution along the statement path is to be suspended until the *call* is "accepted" by the mechanism containing the procedure being called. This rule arises from the fact that only one *call* statement may be in control of a given procedure at any given time and there may be various points in the architecture attempting to *call* the procedure simultaneously. Producing this type of suspension in an S*(QM-1) description would be difficult.

Another construct that produces a mapping problem is the data structures *stack* and *assoc array* that are available in S*A but have no real counterpart in S*(QM-1). It is possible to build a procedure using various S*(QM-1) constructs to simulate the actions of a stack but this would be an unreasonable solution. We will use this construct as an example of one of the major difficulties found in the mapping process. Often there is a construct in the high level language which has no direct counterpart in the constructs of the lower level language. Moreover, the implementation of that construct in the lower level language using a variety of other available constructs proves to be difficult, if not impossible. The lesson to be learned from this example is that the designer should never have used the

*stack* structure at the high design level in the first place. Availability of constructs in the syntax of the design language does not always imply that they will fit into the design process. The *stack* structure was specifically included in the S*A syntax to allow descriptions of architectures where there would be a hardware stack of some sort and thus the mapping would be simple.

So to facilitate the mapping process between languages the designer may have to restrict himself to the use of a subset of the constructs in each language. The final hardware implementation technology that is used may also restrict the use of each of the languages. As an example, including sets of identical processes all executing in parallel, is allowed in S*A architecture descriptions. This would be perfectly reasonable in an architecture which was to be implemented as an array of CPUs, however it is totally absurd if the architecture is to be implemented as microprograms on the QM-1. This illustrates why an awareness of the mapping process is so important.

Thus we have developed a design methodology based on the S* family of languages. We will see how this thesis is a step towards the testing of that design methodology. It includes the development of a simulator, which is a valuable tool, allowing the family of languages concept to be tested with some realistic examples.

# Chapter 5

# Simulation of Computer Architectures

## 5.1 Simulation as part of the design process

It should be noted that simulation of computer architectures is not an exercise that is performed after the design has been completed. Rather, it is to be an integral part of the design process itself. In this chapter we will examine, in some detail, the nature of computer architecture simulations and why they are such useful tools (CAVO81,NUTT78,VAN81,ZUR68).

Why do we need to simulate computer architectures? This question may rightfully be asked given the level of difficulty in producing a simulator for an architecture design. What one should realize is that the purpose of the simulation is much more than to provide the elusive answer to the obvious question "Does it work?". The simulation should provide information to the designer that will help him to evolve his design to reach an optimal solution to the specifications that he is trying to meet.

How can a simulation be so useful? What kind of information should it produce? What are some qualities of a good simulator? These are some of the questions that will hopefully be answered in this chapter.

At the original stages in the design process, the designer should develop a list of specifications that the

final architecture must meet in order to be acceptable. This parallels the specifications that an architect of a building would have during his initial design stages. Within these constraints the designer hopes to achieve an optimal design. The design should be the best possible design that still falls in the acceptable range as laid out in the specifications. In order to achieve this the designer needs some method of measuring the quality of his design relative to certain criteria. This is where the simulation process finds itself the most useful. It allows the designer to choose a set of criteria as being the measure of goodness for the architecture and to simulate the architecture design and evaluate it with respect to those criteria.

This may seem like a fairly straightforward process but often the criteria are very complex. They can also be interrelated, often in an inverse relationship, thus producing the familiar situation where design tradeoffs must be made.

There is a good discussion on one possible set of criteria for evaluating computer architectures in (FULLER77). The basis that was chosen there for the evaluation of computer architectures was basically quantified as three values.

1. S, the number of bytes used to represent a test program.
2. M, the number of bytes transferred between primary memory and the processor during the execution of a test program.

3. R, the number of bytes transferred among internal registers of the processor during the execution of a test program.

These values were measured on a variety of architectures for twelve test programs in order to determine a "best" architecture for a specific application. The choice of the test programs was defined by the application and statistical methods were used to eliminate variations due to programmer style etc. This is one of the few attempts that has been made to provide a quantitative measure of goodness of computer architecture in terms of numerical values. The above set of criteria placed an emphasis on the amount of data transfers that occurs within the architecture as a measure of goodness of that architecture. This is a reasonable approach but is by no means the only criteria that could be used. It can be easily seen though how simulation could be used to evaluate these criteria. Recording the number of data accesses and transfers as part of the simulation process would be a trivial matter.

Let us examine another possible set of criteria that might be used as a basis for the evaluation of a computer architecture. Suppose a computer architecture was being developed that involved an array of processing units, each containing a small local memory and interconnected to the others in some network fashion. One set of criteria for the evaluation of this architecture might be the following.

1. The number of bytes transferred from a CPU to its local

memory during the execution of a test program.

2. The average number of CPU elements that are not doing productive processing at any one time. The term productive processing is used to define all processing other than transferring data between CPU units.
3. The average number of times that a single data item must be transferred between CPU elements before it is involved in a productive calculation.

These criteria place an emphasis on a number of points. The first criterion is aimed at minimizing the number of local data transfers for a single CPU. The second one is aimed at increasing the amount of parallelism in the system as a whole. The final one is aimed at designing an efficient network between the CPUs so as to minimize the non-productive transfer of data between CPU elements. We can readily observe that within this set of criteria design tradeoffs will have to be made. In order to increase the amount of parallelism in this system it may become necessary to do more inter-CPU data transfers to keep all of the CPUs busy. As well, in order to minimize the number of inter-CPU data transfers it may be necessary to temporarily store data items in local memory (rather than send them to adjacent free CPUs) thus increasing the number of local data transfers. In some cases where the local memory size is restrictive, the number of local memory transfers may play a significant role in the design whereas if the local memory size is large this may be a very minor consideration.

With these two sets of examples, we are now better prepared to examine the sorts of things that should be done while simulating an architecture (other than the obvious "execution" of it). In order to provide some results of the simulation, the simulator should include the facility to do a number of things.

1. It is necessary to be able to input data into the simulator to provide the designer with complete control on the input conditions. This can be done in a very simple way by allowing the user to assign values to variable data items at the start and during the simulation. However, if the simulator is to be of any real use at all it would have to include a number of more powerful options. The designer may wish to simulate the execution of an entire operating system on the architecture design and so it would be necessary for the simulator to be able to read in large amounts of data from a source outside of the architecture description. As well, it may be very useful to be able to define variables that take on random values as inputs to the system. These facilities could be built directly into the simulator itself or could be implemented as an application dependent preprocessor, which would prepare sets of commands for input to the simulator.
2. The next major task that the simulator must be able to do is to maintain statistics of various sorts as the simulation progresses. The most basic statistic will

involve the number of accesses for each variable data item as almost all criteria for judging an architecture will involve this in one way or another. The simulator might also maintain statistics of a more sophisticated nature. Complete lists of all previous values could be stored for some variables to allow one to study such things as frequencies of use of certain opcodes in the input data etc. The simulator may produce statistics on average values, maximum and minimum values, deviations and others either continually during simulation or it may produce large amounts of data which can be analyzed after the simulation. Probably the latter technique is the better because, though more costly, it creates a flexibility which is necessary with the varied nature of computer architectures.

3. A breakpoint and restart facility will be needed as it is not usually convenient to run a simulation to its final conclusion before examining its results. Often the designer may wish to interact with the simulator changing values of input variables based on observation of results.

These are some of the things that should exist in an architecture simulator to enhance its usefulness as part of the design process. The diverse nature of computer architectures implies, of course, that it is not unlikely that the designer may want to monitor some aspect of the simulation and the facility to do so will not be in the

simulator. It thus becomes essential for the simulator to be easily expandable. More is implied here than just the writing of well documented, modular code so that someone else can modify the simulator. The simulator should produce as much raw data as possible so that the designer can write his own programs, independent of the simulator, to analyze the data according to his own specific need. It would not be unreasonable to expect the simulator to produce a complete trace of everything that happens at every stage of the simulation with the option given to the user as to which portions he wishes to see. Literature is available on how SLIDE (ALT79) and ISPS (BARB78) have provided many of these facilities. Now let us see how the S*A simulator appears from the user's point of view in the light of these discussions. In chapter 7 we will look at these three simulators in somewhat more detail.

## 5.2 The S*A simulator

This section presents the S*A simulator from the users point of view and to present its strengths and weaknesses in light of the previous discussion.

The first item that one should note is that, though the S*A simulator is completely debugged and running, it is a subset of a complete simulator. A lot of the ideas in the previous section were not implemented in the simulator due to time constraints rather than design decisions. It is

hoped that the simulator will be expanded in the future to include facilities corresponding to these ideas. Thus, any descriptions given here are of the simulator in its current state rather than an expanded future state.

There is a command language for the simulator which provides the interface between the user and the simulator. Whenever the simulator is not executing an architecture description, it is in command mode and is ready to accept commands from the user. A complete description of the command language can be found in Appendix D. It is quite a simple language but as it has been implemented with the language development tools YACC (YACC79) and LEX (LEX79) on UNIX (JOHN80,UNIX79), it is easy to expand as new features are added. Currently the command language allows the user to enter and examine data values in various number bases either one byte at a time or in array ranges, examine **mechanisms** for statistics on their use, initialize statistic counters, continue the simulation of execution or quit.

There is of course work that needs to be done in order to bring the simulator up to production standards but it is a working simulator allowing you to simulate reasonably sophisticated architectures. Some examples of simple descriptions that have been simulated are included in Appendix E. It can be seen from these examples that all of the major constructs, including those involving timing and synchronization as well as the facility for handling contention for the use of **mechanisms**, have been implemented.

This is the major feature of the S*A language and the fact that such constructs as the **case** statement are not available yet is a minor point.

The statistics that are recorded are again currently quite simple. The number of accesses and writes to each **seq** variable is stored and available on request. In addition the number of activations of each mechanism is stored and is also available. These are the only statistics that are currently maintained.

Two simulator commands were implemented into the S*A language itself. The first is the **break** statement which causes control to return to the command mode of the simulator. It also allows the user to provide an optional parenthesized integer value which is displayed on the screen so the user knows at which statement the break point has occurred. There is also a simple one value print statement to display the value of **seq** variables.

There are a number of other minor facilities available to the user. There is an assortment of debug flags which can provide varying amounts of monitoring information as the simulator is running. These are described in Appendix B.

In summary the facility as it stands is complete enough for a designer to simulate small architecture descriptions including relatively complex timing and synchronization processes. This has fulfilled the original goal of the thesis.

# Chapter 6

## Implementation of the S*A Simulator

In this chapter, a more technical description of the design and implementation of the S*A simulator system will be given. Through this description the reader should develop a clearer understanding of the semantics of S*A which, though unambiguously defined, may not always be apparent due to the complex nature of constructs in the language designed to represent interacting parallel events. The reader may also gain insight into how implementation constraints may have had some effect on the design of the simulator system.

The word *system* is used to describe the finished software product as it is really a collection of very distinct software modules. It has been done this way not just to conform to what is considered good programming techniques but as well to provide a basis on which to build applications other than simulation. Simulation is only one of the ways that a user may wish to process an S*A description of an architecture. At the somewhat lower, register transfer level, using ISPS, it has been seen that it is possible to use an architecture description for fault analysis, architecture evaluation, architecture certification and design automation in addition to simulation (BARB80). This concept is no different at the higher architecture level of S*A. Whereas the software currently existing is used only for simulation, the potential uses in other areas have certainly affected the

design of the system.

All of the software for the system has been written in "C", (KERN78) and is currently running on a VAX 11/780. The system is basically composed of a preprocessor, a compiler (including parser) and the simulator.

## 6.1 Preprocessor

The preprocessor receives as its input the S*A description of an architecture system which the user wishes to simulate. Presently, it performs the task of expanding homogeneous sets of mechanisms whereby the user can describe a large number of identical mechanisms by only providing a single mechanism description and an expression to define a set of mechanisms based on that description. The exact operation of the preprocessor is described in Appendix A and for S*A descriptions that do not include homogeneous sets of mechanisms, it can be left out.

## 6.2 Compiler

The compiler is the next major component of the system. It will take as its input the output of the preprocessor, if it is used, otherwise, a user written S*A description. The compiler was written using the compiler-compiler YACC (YACC79), one of the tools provided with the UNIX operating system. YACC takes a description of the grammer for S*A and produces the C code for a parser for the language. It also

allows the designer to give blocks of C code to be executed upon recognition of certain rules in the grammer. This allows one to design and build a relatively complex compiler in a reasonably short period of time. LEX (LEX79) is used to produce the lexical analyzer which accompanies the parser. LEX takes a listing of rules provided by the designer and produces C code for a lexical analyzer which will read in characters and recognize keywords, identifiers, etc., passing lexical tokens on to YACC. The lexical analyzer also produces a listing of the tokens that have been produced for debugging purposes (see Appendix B). The compiler will read in the S*A code and build a parse tree of the S*A code in memory. It provides all of its own memory management including such things as maintaining a table to store identifier names, etc.

### 6.3 Parse tree

The parse tree that is produced is a compact form of the compiled S*A description. It is this form that is written to a file and later used as the basis for the simulation of the architecture. This representation of the architecture could also be used for some of the other applications mentioned earlier. It is for these reasons that the parse tree is quite important and thus a fairly detailed informal description of it will be given even though it is invisible to the user of the simulator system.

The parse tree is composed of a number of nodes connected together in tree like fashion. It is not a true tree as it is possible for cycles to occur. The different types of nodes can be classified into a number of categories.

1. Header nodes

   Each system, mechanism, private procedure, public procedure and function has a header node. This node contains pointers to all other nodes which contain associated information for the particular entity. For example, a system header node contains fields for, among other things, pointers to:

   a. the list of header nodes for mechanisms contained within the system.
   b. the list of header nodes for other systems contained within this system.
   c. the list of global variables declared within this system.
   d. the list of private variables declared within this system
   e. the list of public procedure references that occur within this system.
   f. the list of global variable references that occur within the system.
   g. the identifier table, specifically to the id name for the system.

The system header is active only during compilation, in that once the parse tree is completed it is not used again by the simulator. Some of the other header nodes such as the mechanism header node are of a more functional nature, in that they store information such as lists of awaiting procedure calls that will dynamically change during simulation.

2. Declaration nodes

Another category of nodes is the set of declaration nodes. Each variable that is declared is represented by a declaration node. These declaration nodes are organized in a hierarchical fashion matching their organization in the S*A description. The **seq** node is the elementary unit of all declarations. The **bit** declaration is considered simply as a **seq** declaration of length one. **Tuples**, **arrays**, **stacks**, and **assoc** arrays are represented by a declaration header node which contains pointers to the declaration nodes for all variables contained under them. **Seq** declaration nodes that occur within an **array**, **stack**, **tuple**, or **assoc** array will contain the value of their displacement within that higher level structure relative to the other declarations under the higher level structure.

These declaration nodes are mapped into positions in an imaginary memory known as the BASE. Each declared variable of elementary type **seq** will have 1 position in the BASE. For variables declared as **array**, a block of

positions in BASE is reserved while the actual position to be used for a given variable reference cannot be determined until simulation time when the value of the subscripts can be determined.

All references to variables within the parse tree are eventually resolved by the compiler by means of a pointer to the declaration node for that variable. The simulator will then extract the position in the BASE which represents that particular **seq** variable (using such information as the current value of subscripts if it is an array). At simulation time, the imaginary BASE memory is realized as an array containing, in each position, such information as the current value of that variable and a number of statistics about the variable.

3. Statement nodes

Another category of nodes is the set of statement nodes. Each executable statement in the S*A code is represented by a statement node, the exact type of which depends on the type of statement involved. Contained in the statement node are pointers to the various components of the statement and a field indicating the type of statement that it is. The node also contains a flag indicating the current execution status of the statement. This flag is dynamically changed during simulation to monitor the status of the statement. This is used for such things as controlling the sequential execution of statements by not allowing one statement to

execute until another statement has completed execution.

4. Other node categories

There are a number of other, less important types of nodes in the parse tree. Some of these are of a temporary nature being used to pass sets of values from one part of the compiler to another.

When the building of the parse tree is completed, and all references to variables, functions, and procedures have been resolved to point to their respective header nodes, the parse tree along with all other necessary information is written to a file and is later used by the simulator as the basis of the simulation. It is expected that this parse tree would also be used by other applications other than simulation.

## 6.4 Simulator

The simulator is the final major software component of the system. The ideas in (ZEIG76,ZEIG78) aided the author in designing the simulator. It consists of a parser, again produced using YACC and LEX, to recognize the command language that the user employs, as well as code which simulates the execution of the parse tree which it receives from the compiler. By keeping the parser of the simulator command language separate from the code used for simulation, it becomes easy to suspend the process of simulation to

accept commands from the user.

The following basic technique was used to handle the problems involved with simulating parallel statements as well as **call** and **act** statements to procedures in other mechanisms.

Each mechanism contains in its header node a dynamically changing list of all statements contained within the mechanism that are currently executing. For example, a parallel statement is considered to be executing until all of its substatements have finished execution. All of its substatements are placed onto the executing list and can be in execution state concurrently. In the case of sequential statments however, when one statement finishes executing and comes off the list, the next statement begins executing and is placed onto the list. The simulator goes through each executing statement in each active mechanism, and executes it for one time step. The exact definition of one time step depends on the type of statement. The operations involved in the execution of a statement for one time step may include the assigning of a value to a variable as is the case with the assignment statement, or it may involve the evaluation of a predicate and possible initiation of another statement for execution as is the case with the **if** statement.

This round robin technique for executing statements produces, from the users point of view, a random execution of parallel paths in the S*A description. One of the interesting results of this technique occurs in the

semantics of the **sig** and **await** statements. If a number of **await** statments are waiting on a specific synchronization variable, and a **sig** statement is applied to that variable, it is not readily apparent as to which **await** statement will be the first to continue executing. One cannot assume that the first await statment to have executed will be the first to continue execution. Whichever **await** statement is the first to be executed by the round robin technique after the **sig** statement will be the one that is the first to complete.

The calling and activating of public procedures is a relatively complicated event in the simulation. Each mechanism header node contains a pointer to a list of all **call** or **activate** statements that are currently requesting the exclusive control of the mechanism. The reader should remember that the calling or activating of any procedure always requires exclusive control of the mechanism in which it is contained. If a particular mechanism is currently inactive, the simulator will check the list on the mechanism header node and if there are any requests waiting for control of the mechanism, it will initialize the mechanism. This will involve fetching and passing all of the arguments in the call statement to the parameters of the particular procedure that is being called. When all statements in a mechanism have finished executing, the mechanism is deactivated. This may involve passing back output parameters to the calling statement. It may also be necessary to change the status of the calling statement to allow it to continue

executing (see Appendix C). As well the mechanism is flagged as inactive so that the simulator will initialize the next waiting call statement if there is one, on its next round robin pass through this mechanism.

There are a number of disadvantages to the technique of round robin or polling of mechanisms and statements. The most obvious is the lack of efficiency. It can become expensive to continuously examine all executing statements within an S*A description to see if they can resume execution. This cost becomes more apparent when one realizes that each parallel or sequential statement separator in the S*A description is represented as a statement in the parse tree which begins executing as soon as one of its component statements commences execution, and remains in the execution state until both of its component statements have terminated execution. As well, an **await** statement is also continuously executing while it is waiting on the value of a synchronization variable. It would have been more efficient to have used a type of interrupt driven technique where **await** statements would be awakened and restarted when a **sig** statement occurs. In addition, control paths might have been followed until they were suspended by a timing constraint, rather than only execute for one time step at a time.

The prime reason that these ideas were not implemented was because of the added complexity involved. The round robin technique, though somewhat inefficient, is a very straightforward and simple method to implement. It also

produces a more evenly distributed parallelism with all simultaneously executing control paths progressing at approximately the same rate. This type of simulation would appear to have more semblance of the real time parallelism that would occur in a computer system.

The final point that will be discussed in this chapter involves the storage of the values of variables in the BASE memory array. Each **seq** variable is mapped to and stored in a 32 bit integer variable in C. Thus, it can take on negative as well as positive values and arithmetic operations will produce results that would occur if the same arithmetic operation were performed on an integer variable in C. Any operations performed on portions of a **seq** variable are done by first extracting the required bits (via masking) from the operands and placing them into temporary C integer variables. The operation is then performed and the result is placed into the appropiate bit positions of the result variable (again via masking). If the result contains too many significant bits for the number of available bit positions then truncation at the left occurs and significant bits may be lost. The user should be aware of this as no error messages result when truncation occurs. One point to note is that if there is a complex expression on the right hand side of an assignment statement, the truncation of bits to fit the result does not occur until the entire expression has been evaluated and the assignment is about to occur. Furthermore, for each individual operation, the inherent

length of the result is the maximum length of the two operands. We can see that the conspicuous constraint that this particular implementation results in is that **seq** variables are limited to 32 bits. Moreover, the user must concern himself with the retention of significant bits when he starts to approach that limit where complex operations are involved.

An attempt has been made in this chapter to explore some of the design decisions that were made in creating the simulator system and how they have affected the final product. It is hoped that the reader now has a clearer understanding of how the simulator works as well as the rationale behind its general design.

# Chapter 7

## A Comparison of Three Simulators

This chapter provides a brief comparison of three different simulators currently implemented for simulating computer architectures. The first is a simulator for architecture descriptions written in S*A, which forms a large part of this thesis. This is compared with simulators for architecture descriptions written in ISPS (BARB78), and SLIDE (ALT79).

All three simulators are designed for architecture descriptions in a specific description language. As well, all these description languages are for architecture descriptions at or near the register transfer level. This chapter will not deal with the languages themselves, but with only the simulators, both from a users viewpoint as well as a technical viewpoint. A brief discussion of these and other architecture description languages can be found in chapter 3.

### 7.1 User's Viewpoint

Of the three simulators we have chosen to study, ISPS has been in existence for the longest time and is by far the most complete. For this reason we will briefly review its major features and then compare the other simulators to it. The ISPS simulator includes the following facilities.

1. the ability for the user to choose between a number of

different sets of arithmetic operations such as two's complement, one's complement, unsigned binary, etc. Different number bases can also be used for entering and reading values.

2. a facility for the user to define certain processes to contain critical sections (i.e. they can only be activated from one calling statement at a time).
3. a facility to save the current state of a simulation for continuation at a later time.
4. a facility whereby certain predefined variables and procedures are available to the designer in his architecture description.
5. a facility to "break" out of possible infinite loops into command mode.
6. a fairly sophisticated breakpoint facility to put the user into command mode when certain conditions, based on process activation, variable accesses, statement execution, or loop counters become true.
7. a facility to associate times with statements and operations and to monitor total times of runs or parts of runs.
8. a facility to trace the simulation.
9. a facility to set and interrogate the values of variables, either individually or in arrays.
10. the ability to keep fairly extensive statistics on variables for examination upon completion of a simulation run.

11. Various other facilities such as simple symbolic substitution in the command language, the ability to include external files in the command file, a news facility for current changes, a help facility and others.

In short, the ISPS simulator has been installed within the framework of a fairly complete design environment.

The S*A simulator has some of the same features as the ISPS simulator but being more in its infancy, it does not include such a sophisticated design environment. Some of its features are the following.

1. a simple breakpoint facility for terminating the simulation of the architecture description and putting the user into command mode.
2. a simple facility for printing of variable values during simulation.
3. a facility to set and interrogate both **seq** and **array** type variables.
4. a small number of statistics are maintained on each variable, mechanism and procedure.
5. the radix of values entered and read may be changed (i.e octal,decimal,binary or hex).

Many of the features available in the ISPS simulator are presently missing in the S*A simulator but this is primarily due to time constraints. The S*A simulator has been designed with future expansion as a primary consideration. One difference between the two is that the

S*A simulator does not include facilities for associating times with statements and monitoring the simulated execution times of runs. This is because of the difference in the architecture level that the two languages were designed for. S*A was designed for higher architecture level descriptions where one is typically not concerned with low level timing entities such as clock pulses or seconds.

The SLIDE simulator is also a reasonably simple simulator. The available facilities one finds in it include the ability to trace the execution of the simulation, the ability to set up sets of connections between components in the architecture descriptions and the ability to save these sets of connections for retrieval at a later time. The simulator can be run for a specified number of time units. The SLIDE simulator is still in a developmental stage and we expect it to include many more facilities in the future.

In summary, ISPS is the only computer architecture simulator of the three that could be considered *production* quality but the others are at least running and usable.

## 7.2 Technical Viewpoint

As all three of the simulators are reasonably complex, we will not go into too much detail in this section. For a more detailed technical description of S*A see chapter 6, while the ISPS and the SLIDE simulators are described in detail in (BARB78) and (ALT79) respectively.

The most interesting aspect of the simulators is how they handle parallelism both within and between procedures. S*A and ISPS are somewhat similar in this aspect. Both employ a round robin technique for all statements that are currently running in parallel at any one time. The major difference between the two is that in ISPS each statement is executed completely whereas in S*A each statement is run for only one *timestep*. A *timestep* will be defined for each type of statement and it may or may not involve the execution of the entire statement. Also included in a *timestep* may be the testing of a set of conditions based on the values of varibles in the architecture description. It is not always known until simulation time the number of timesteps required to complete the execution of a particular statement. The reason for this is that in S*A there are higher level constructs such as **sig** and **await** which must continually monitor a variable until a certain condition becomes true, and thus cannot always execute to completion at any particular time.

The S*A and ISPS simulators are also quite similar in how they handle inter-procedure parallelism. Both will queue calls to a procedure from different sources and run them one at a time. The main difference is that in S*A a procedure call is suspended until the procedure is free and allocated to the call statement (see Appendix C).

SLIDE is a little different from the above two languages in its handling of parallelism. It will map

descriptions of an architecture in SLIDE onto code written in SIMULA (BIRT73), a language designed for simulating concurrent events. SIMULA contains a scheduler which will then solve the problems in scheduling parallel events during the simulation. It is convenient to have an available facility to handle the scheduling problems but the mapping between the SLIDE and SIMULA descriptions is not trivial. Parallel statement streams result in the creation of a new SLIDE process to handle the execution of the second path.

All of the simulators involve the use of an intermediate data structure between the compilation and simulation stages. Both ISPS and SLIDE use a similar character representation as they belong to the same multi-level simulation facility. In the S*A project it was realized that simulation is only one application of computer description languages (BARB80), and thus it is necessary to have a compact intermediate form for S*A descriptions. Whereas in ISPS these other possible applications for architecture description languages have, to some degree, been realized, SLIDE and S*A are both still in a developmental stage working towards more complete production systems.

# Chapter 8

## Conclusion

In closing, let us examine what has been accomplished by this thesis and the new directions that these accomplishments now point to.

The *family of languages* is the basis of a methodology for the design, testing and verification of computer architectures. It is based on the use of symbolic languages for the representation of computer architectures during the design process. Specifically, it involves the use of a number of related languages, each designed to represent the architecture at a particular level of abstraction. In order to understand how each language is designed for a specific level of architecture descriptions, it was necessary to study the nature of levels of abstraction as it relates to computer architecture descriptions. It was also necessary to look at how these levels of abstraction relate to the specific language being used to represent them and to introduce the concept of a *mapping* between the level of abstraction and the language being used. Hopefully, the reader is now more aware of what a good mapping consists of and how it can affect the design.

With the above information, and a brief discussion on a number of current computer architecture description languages, it was then possible to take a closer look at the design process itself. We have seen that the family of languages approach involves an iterative design process. One

of the steps in this process is the simulation of the computer architecture based on its description in an appropiate language. We have seen the importance of this step and thus one of the specific goals of this thesis was to develop and implement a software compiler and simulator for the language S*A of the S* family. This simulator was compared to a number of other simulators of architecture description languages in order to give the reader an idea of its strengths and weaknesses.

The future course of this project is now more clear. It requires the use of the simulator for S*A and some of the ideas presented in this thesis in "real" applications. This will allow us to pragmatically measure the effectiveness of the family of languages approach to architecture design. It will also allow the incorporation of new ideas into the design process involved with this approach. Hopefully this will be a step towards the goal of a *design discipline* that was introduced at the beginning of this thesis.

## Bibliography


ALT79 - Altman,A.H., "The SLIDE Simulator: A design and evaluation tool for I/O and interfacing strategies", *Masters Research Project Report*, Dept. of Electrical Engineering, Carnegie-Mellon University, Dec 2,1979.

APL74 - Gilman,L. ,Rose,A.J., *APL: An Interactive Approach*, second revised printing, John Wiley and Sons inc., 1976.

BARB75 - Barbacci,M.R., "A Comparison of Register Transfer Languages for Describing Computers and Digital Systems", *IEEE Transactions on Computers*, Vol. c-24, No. 2, Feb. 1975.

BARB77a - Barbacci,M.R., Siewiorek,D., Gorden,R., Howbrigg,R., Zuckerman,S., "An Architectural Research Facility - ISP descriptions, simulation, data collection", *AFIPS Conference Proceedings*, Vol. 46, National Computer Conference, 1977.

BARB77b - Barbacci,M.R., Barnes,G.E., Cattel,R.G., Siewiorek,D.P., "The ISPS Computer Description Language", *The symbolic manipulation of computer descriptions*, Department of Computer Science, Carnegie-Mellon University, 1977.

BARB78 - Barbacci,M.R., Nagle,A.W., "An ISPS Simulator", *The symbolic manipulation of computer descriptions, ISPS Application Note*, Department of Computer Science, Carnegie-Mellon University, March 7, 1978.

BARB80 - Barbacci,M.R., Northcutt,J.D., "Applications of ISPS, an Architecture Description Language", *Journal of Digital Systems*, Vol IV, Issue 3, 1980.

BIRT73 - Birtwistle,G., et al., *SIMULA BEGIN*, Petrocelli/Charter, 641 Lexington Ave, N.Y.C, N.Y., 10022, 1973.

BREUR75 - Breuer,M.A. (editor), *Digital System Design Automation: Language, Simulation, and Data Base*, Computer Science Press inc., Woodland Hills, California, 1975.

CAVO81 - Cavouras,J.C., Davis,R.H., "Simulation Tools in Computer System Design Methodologies", *The Computer Journal*, Vol 24, No. 1, 1981.

CHU65 - Chu,Y., "An Algol-like Computer Design Language", *Communications, A.C.M.*, Vol 8, pp. 607-615, Oct. 1965.

DASG78 - Dasgupta,S., "Towards a Microprogramming Language Schema", *Proceedings MICRO-11*, 1978.

DASG80 - Dasgupta,S., "Some Aspects of High Level Microprogramming", *A.C.M. Computing Surveys*, Vol. 12, No. 3, Sept. 1980.

DASG81a - Dasgupta,S., Olafsson,M., "Towards a Family of Languages for the Design and Implementation of Machine Architectures", *Technical Report TR81-5*, Department of Computing Science, University of Alberta, June 1981.

DASG81b - Dasgupta,S., "S*A: A Language for Describing Computer Architectures", *Proceedings of the IFIP TC-10 Fifth International Conference on Computer Hardware Description Languages and their Applications*, Sept 1981.

DULEY68 - Duley,J.R., Dietmeyer,D.L., "A Digital System Design Language (DDL)", *IEEE Transactions on Computing*, Vol C-17, pp. 850-861, Sept. 1968.

FULLER77 - Fuller,S.H., Shaman,P., Lamb,D., "Evaluation of Computer Architectures via Test Programs", *AFIPS Conference Proceedings*, Vol 46, National Computer Conference, 1977.

JOHN80 - Johnson,S.C., "Language Development Tools on the UNIX System", *IEEE Computer*, Aug. 1980.

KERN78 - Kernighan,B.W., Ritchie,D.M., *The C Programming Language*, Prentice-Hall Inc., New Jersey, 1978.

KLASS81a - Klassen,A.B., Dasgupta,S., "S*(QM-1): An Instantiation of the High Level Microprogramming Schema S* for the Nanodata QM-1", *Technical report TR81-4*, Department of Computing Science, University of Alberta, May 1981.

KLASS81b - Klassen,A.B., "S*(QM-1): An Experimental Evaluation of the High Level Microprogramming Language Schema S* Using the Nanodata QM-1", *Masters Thesis*, Department of Computing Science, University of Alberta, 1981.

LEUNG79 - Leung,C.K.C., "ADL: An Architecture Description Language for Packet Communication Systems", *Computer Structures Group, Memo 185*, M.I.T., Oct 1979.

LEX79 - Lesk,M.E., Schmidt,E., "LEX - A Lexical Analyzer Generator", *Unix Programmer's Manual*, seventh edition, Vol 2A, Jan 1979.

NANO77 - Nanodata Corporation, *QM-1 Hardware User's Manual*, Third Edition, Revision 1, Buffalo, New York: Nanodata Corporation, 1979.

NUTT78 - Nutt,G.J., "A Case Study of Simulation as a Computer System Design Tool", *IEEE Computer*, Oct. 1978.

PARN72 - Parnas,D.L., "A Technique for Software Module Specifications with Examples", *Communications of A.C.M.*, Vol. 15, No. 5, May 1972.

PARK81 - Parker,A.C., Wallace,J.J., "SLIDE: An I/O Hardware Description Language", *IEEE Transactions on Computers*, Vol. C-30, No. 6, June 1981.

UNIX79 - Bell Labs, *UNIX Programmer's Manual*, seventh edition, Vol 2A, Jan 1979.

VAN81 - Van Dam,A., Barbacci,M.R., Halatsis,C., Joosten,J., Letheren,M., "Simulation of a Horizontal Bit-sliced Processor Using the ISPS Architecture Simulation Facility", *IEEE Transactions on Computers*, Vol. C-30, No. 7, July 1981.

YACC79 - Johnston,S.C., "YACC: Yet Another Compiler-Compiler", *UNIX Programmer's Manual*, seventh edition, Vol 2A, Jan 1979.

ZUR68 - Zurcher,F.W., Randell,B., "Iterative Multi-level Modelling, a Methodology for Computer System Design", *Proceedings of IFIP Congress*, Information Processing 68, Volume 2, 1968.

ZEIG76 - Zeigler,B.P., *Theory of Modelling and Simulation*, John Wiley and Sons, 1976.

ZEIG78 - Zeigler,B.P., et al., *Symposium on Modelling and Simulation Methodology*, Weizman Institute of Science, North Holland Publishing Co., 1978.

## Appendix A
## The Preprocessor

The basic function of the preprocessor is to expand the shorthand notation for writing sets of identical mechanisms. It is merely a symbolic substitution facility. Basically it accepts input of the form

**typemech** *mechid* (- - -) **endtypemech**

.

.

**set** (i..j) of *mechid*

where "i" and "j" are integers and "(- - -)" is a mechanism description.

It will produce corresponding output of the form

**mech** #i *mechid* (- - -) **endmech**

**mech** #i+1 *mechid* (- - -) **endmech**

.

.

**mech** #j *mechid* (- - -) **endmech**

Any number of pairs of **typemech** and **set** expressions may occur in the input but they must be matched (i.e. the corresponding **set** expression for each **typemech** expression must occur after it and before any others. Furthermore, the *mechid*s must match). The keywords **set** and **typemech** must not be used in any other context. The preprocessing facility, though presently very simple, allows for future expansion where it may be used to implement a general symbolic substitution facility and a macro facility.

## Appendix B
## Debugging Aids

This appendix provides a brief outline of some of the debugging aids currently existing in the S*A simulator system. This information may prove useful to anyone who may wish to expand or modify the existing software system. It is also useful if at a time in the future, bugs are uncovered that have, as yet, gone unnoticed.

One of the difficulties in debugging this system is that when the parser is run it returns a 1 or 0 depending on whether it has accepted the input as a grammatically correct S*A description or not. To aid the user in determining exactly where a syntax error has occurred, the following facility is available. In the S*A description the keyword "%LEXDEBUG=ON" will activate the lexical debugging facility which will print into the file "./lex.debug" information on each lexical token it encounters. Included in this information are the line numbers of the original source file. The keyword "%LEXDEBUG=OFF" will turn this facility off. The facility can be turned off and on as many times as one wishes within a single S*A description. This will enable the user to pinpoint which symbol was being processed when a syntax error occurs.

Within the compiler there are a number of debugging flags which control the printing of debugging information during compiler execution. There value is set in the routine "init" found in file "init.c" under the directory

"/mnt/grad/makaren/thesis/parse". During normal execution of the compiler, these flags are assigned the value "DBOFF". To activate the printing of debugging information, they must be instead, assigned the value "DBON" and then the compiler must be recompiled (use "make" under the above directory). The three flags are

1. basedb - causes the printing of debugging information concerning the positions of declared variables in the imaginary BASE memory (see chapter 6).
2. sysdb - causes the printing of debugging information available at the end of each system and mechanism encountered. This will include lists of unresolved variable references, etc.
3. misc1db - causes the printing of various other debugging information (it can be quite lengthy) as the compiler executes.

There are four similar debugging flags in the simulator as well. Again, they are all initially set to the value "DBOFF" during normal execution. To activate the printing facility, they must be instead assigned the value "DBON". This must be done in the routine "init" located in the file "init.c" under the directory "/mnt/grad/makaren/thesis/sim". The four flags are as follows.

1. mechdb - causes information about which mechanisms are currently active and the number of pending procedure calls on each mechanism.
2. initdb - causes information during the initialization

phase of simulation to be printed including the names of all variables in the S*A description as they are placed in the hash table.

3. stepdb - causes information concerning the statements in the S*A description currently being executed to be printed.
4. misc2db - causes various other information that may be useful during debugging to be printed. Note that it may be lengthy.

The last point is in regards to the error messages that are produced by the S*A simulator system. The standard format for error messages produced during compilation is

**ERROR**-*NNN*-com, in *filename*, *message*

where *NNN* is a unique error number, *filename* is the name of the file containing the routine that was executing when the error occurred and *message* is the error message indicating the nature of the error.

Similarly, error messages produced during simulation have the standard form

**ERROR**-*NNN*-sim, in *filename*, *message*

where *NNN*, *filename* and *message* are defined as above.

When an error occurs, the user can scan the *filename* for the unique error number to quickly locate the place where the error occurred.

## Appendix C
## Subroutine Calls

This appendix will provide a discussion on the different types of subroutine calls. This discussion will deal with some of the various ways that a subroutine call construct can be implemented and the rationale for the particular implementation used in the S*A simulator.

When a subroutine is called, there is a waiting time (possibly 0) before the routine is available and allocated to the particular calling statement. Parameters may be passed from the main routine to the subroutine at the time of allocation. The main routine may or may not continue executing while waiting for this allocation to occur. As well, when a subroutine is finished executing, it may have values to return to the mainline routine. The main routine may or may not wait until completion of the subroutine before continuing execution.

Based on the above criteria, subroutines can be classified into 4 types.

Figures 1-4 illustrate these four types of subroutine calls. The legends are as follows.

SW = subroutine waiting,

SX = subroutine executing,

MW = mainline waiting and

MX = mainline executing.

1. Type 1 (see Fig. 1). The main routine does *not* wait for allocation of the subroutine and does *not* wait for

completion of the subroutine before continuing execution. No direct parameters are passed to the subroutine though values may be passed via dedicated global variables. Typically, the subroutine does not return any values upon completion.

2. Type 2 (see Fig. 2). The main routine does *not* wait for allocation of the subroutine but it *does* wait for completion of the subroutine before continuing execution. Again, there are no direct parameters passed but values can be returned upon completion.
3. Type 3 (see Fig. 3). The main routine *does* wait for allocation of the subroutine but does *not* wait for subroutine completion before continuing execution. Since the mainline waits for allocation of the subroutine, parameters can be passed to it. However, no values are returned to the mainline when the subroutine completes.
4. Type 4 (see Fig 4). The main routine waits for *both* allocation and completion of the subroutine before continuing execution. This is the common type seen in most conventional programming languages and allow for both parameter passing and values to be returned.

The semantics of S*A are such that only subroutine calls of types 3 and 4 are implemented directly in the language. This can be done using the **act** and **call** statements respectively. There is some question as to whether S*A should implement type 1 and 2 subroutine calls. Specifically, the question is whether to allow the main

routine to continue executing before the subroutine is allocated. In S*A allocation of a subroutine corresponds to activation of the mechanism containing the subroutine. The choice that was made *not* to allow the mainline to continue executing until the subroutine has been allocated. The reasons behind this choice were as follows.

- This allows for passing of parameters on all types of subroutine calls (both **act** and **call** ), which is a useful facility.
- This will prevent certain types of errors from stacking up an infinite number of subroutine call requests.
- There are very few hardware components that operate in a fashion where they can be activated from a number of different spots, with no acknowledgement given to the caller.

It should be noted though that type 1 and 2 subroutine calls can be implemented in S*A using the semaphore constructs. It is not however a trivial implementation.

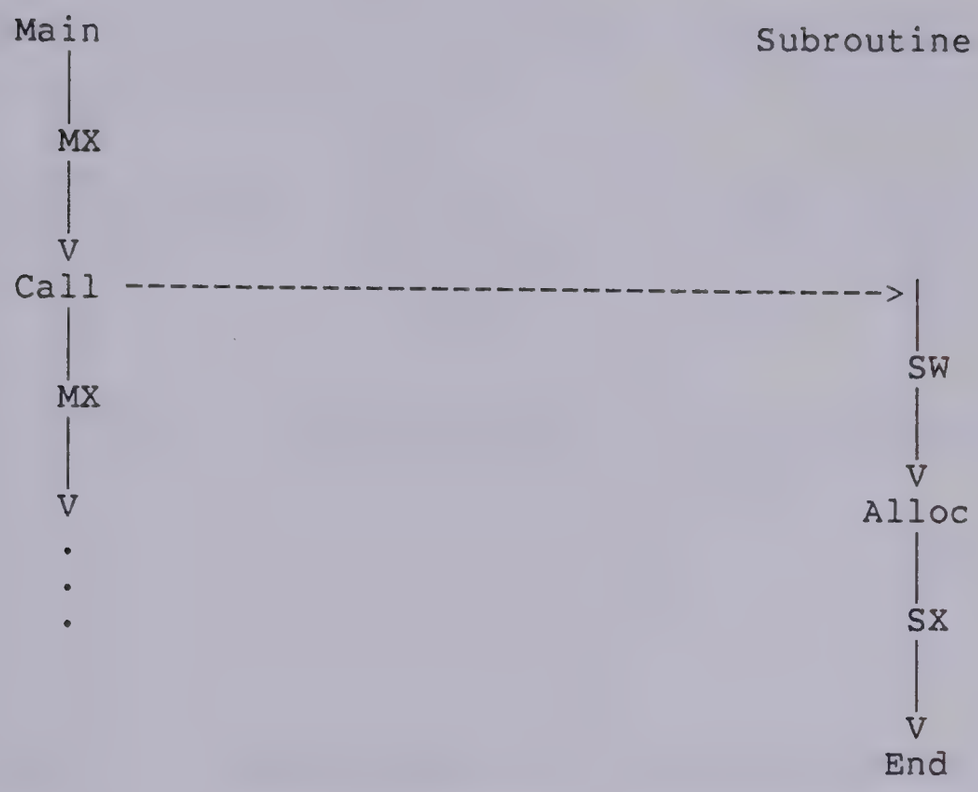


Fig. 1 - Type 1, No Waiting for Allocation or Completion

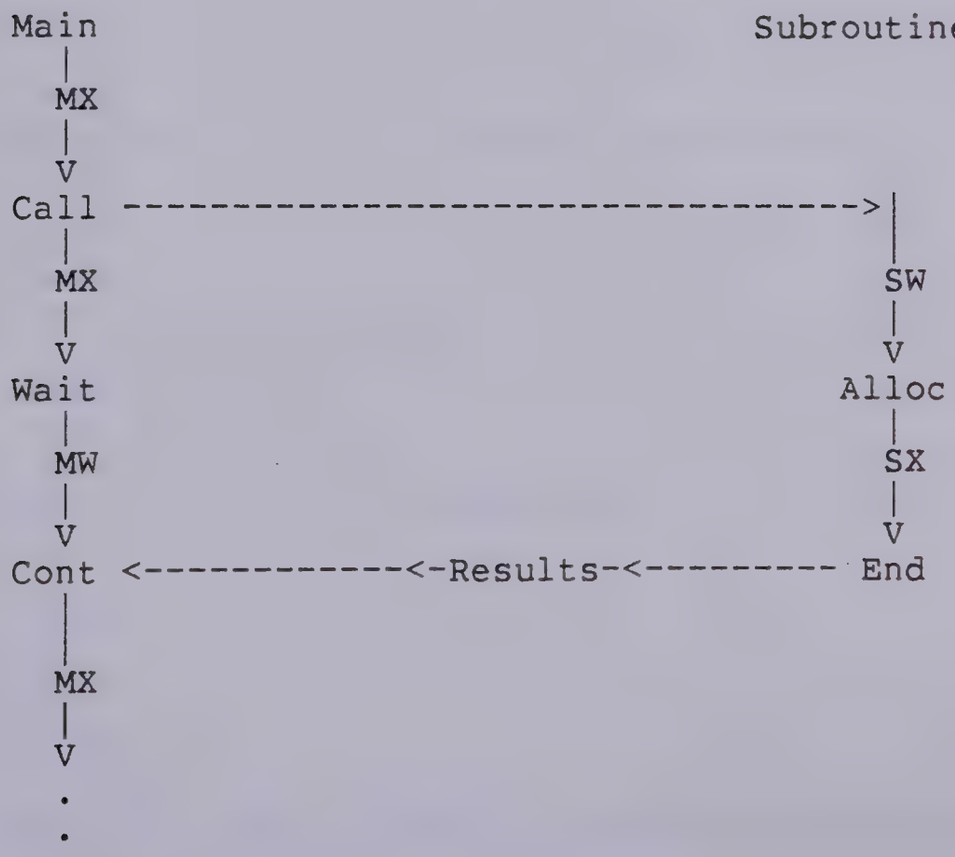


Fig. 2 - Type 2, Waiting for Completion Only

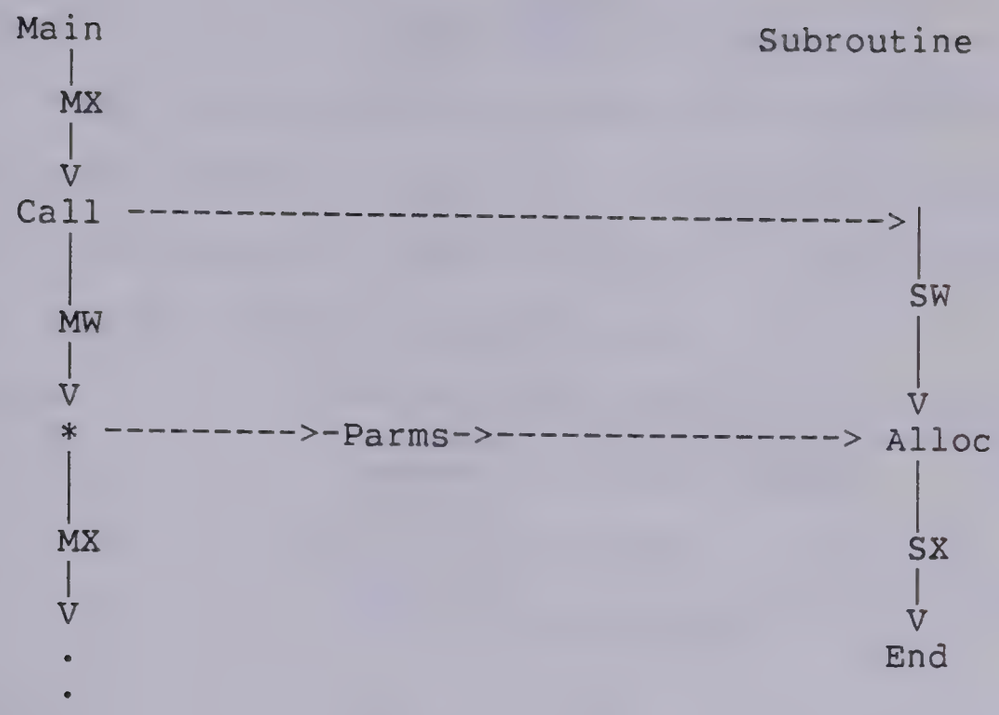


Fig. 3 - Type 3, Waiting for Allocation Only

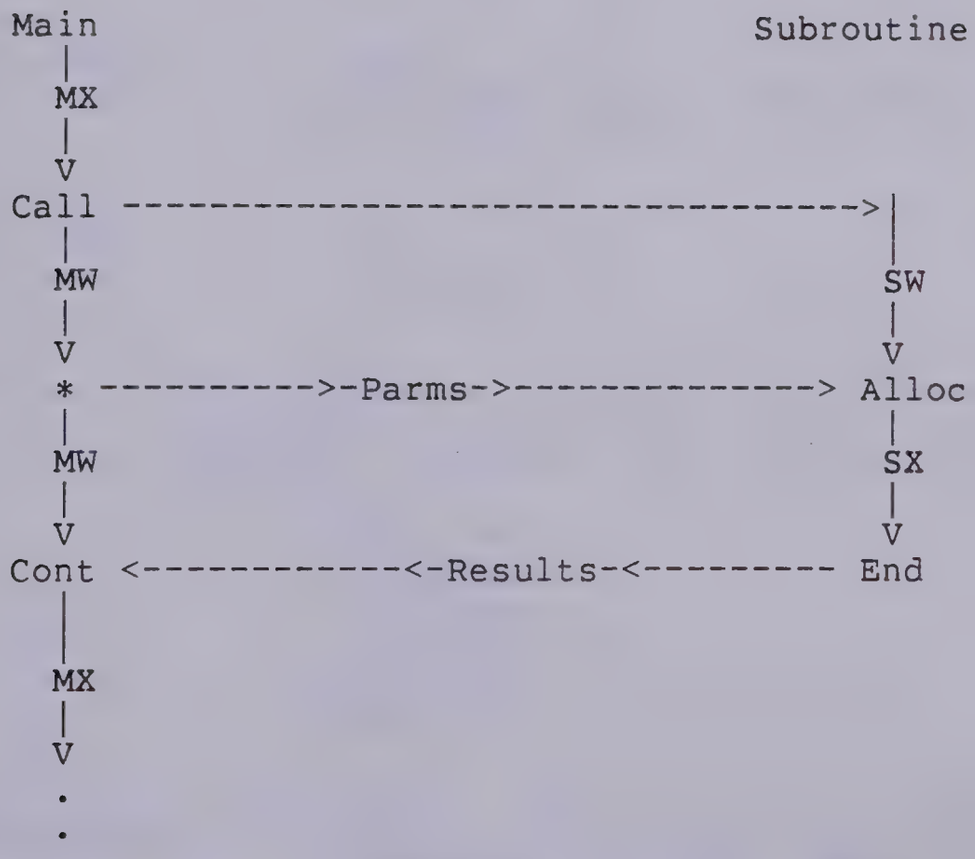


Fig. 4 - Type 4, Waiting for Allocation and Completion

## Appendix D
## The Command Language

The following is the grammar for the current command language for the simulator for architecture descriptions in S*A. All expressions delimited by /* .. */ are comments and are for the reader's benefit only.

```
output          /* The user may enter one or more */
                /* statements terminated by a GO  */
                /* command or a QUIT command      */
                  : go_stmt
                  | stmt_set go_stmt
                  | quit_stmt
                  | stmt_set quit_stmt
                  ;

go_stmt           : GO
                  /* The GO command resumes execution */
                  /* of the simulator from the last   */
                  /* breakpoint.                      */
                  ;

quit_stmt         : QUIT
                  /* The QUIT command terminates      */
                  /* the simulation                   */
                  ;

stmt_set          : stmt
                  | stmt_set stmt
                  ;

stmt     /* The following are the possible statement */
         /* types which the user may enter.  Notice  */
         /* that all statements are terminated by a  */
         /* semicolon                                */
                : rep_stmt ';'
                | write_stmt ';'
                | read_stmt ';'
                | init_stmt ';'
                | mech_stmt ';'
                | debug_stmt ';'
                ;

debug_stmt        : DEBUG    /* Sets all debug flags ON  */
                  | DEBUGOFF /* Sets all debug flags OFF */
                  /* The debug flags are those specified */
                  /* in Appendix B                       */
                  ;
```

```
mech_stmt       : MECH ID
          /* Prints out stats on the mechanism "ID"    */
                | MECH SLASHALL
          /* prints out stats on all of the mechanisms */
                ;

rep_stmt        :
               /* Sets the default number base to be    */
               /* used for printing values of variables */
                : REP '=' HEX
                | REP '=' BIN
                | REP '=' OCT
                | REP '=' DEC
                ;

init_stmt       : INIT
                /* initializes all variables to zero */
                /* and initializes all statistics    */
                ;

write_stmt      : WRITE id rep_cl number
                /* Assign the value "number" to the */
                /* seq variable "id".  The value is */
                /* interpreted in the number base   */
                /* given in "rep_cl" or if not      */
                /* present, the default base.       */

                | write_array
                /* used for writing arrays of values */
                ;

write_array     : WRITE ID '(' integer ':' integer ')'
                                    rep_cl number
                | write_array ','  number
              /* Assign the values given as a list of */
              /* "numbers" to the array "ID", in the  */
              /* positions specified by               */
              /* "integer : integer ". The values are */
              /* interpreted to be in the number base */
              /* specified by rep_cl, or if not       */
              /* present, the default base.           */
                ;

read_stmt       : READ id rep_cl value_cl stat_cl
          /* Print the value and/or stats           */
          /* (as specified by value_cl and stat_cl) */
          /* of the variable "id", in the base      */
          /* specified by rep_cl, or if not present */
          /* the default base */

                | READ ID '(' integer ':' integer ')'
                                     rep_cl number
          /* print the value and/or stats of the items    */
          /* in the array "id" in the positions specified */
```

```
          /* by "integer : integer".                          */

              | READ SLASHALL rep_cl value_cl stat_cl
          /* print values and/or stats on all variables */
          /* in this simulation run                           */
              ;

rep_cl        :
            /* sets the number base to be used for */
            /* the printing of values              */
              :  HEX
              |  BIN
              |  OCT
              |  DEC
              |  /* null implies default */
              ;

value_cl      :
              | VALUE  /* print value(s) of     */
                       /* specified variable(s) */
              ;

stat_cl       :
              | STATS  /* print stats on         */
                       /* specified variable(s)  */
              ;

id            : ID                    /* seq variable  */
              | ID '(' integer ')' /* array element */
              ;

number        : digit
              | number digit
              ;

    /* the following rules interact with the   */
    /* lexical analyzer to interpret integers  */

integer       : dec_digit
              | integer dec_digit
              ;

dec_digit     : DIG_8_TO_9
              | DIG_2_TO_7
              | DIG_0_TO_1
              ;

digit         : DIG_A_TO_F
              | dec_digit
              ;
```

## Appendix E
## Examples

In this appendix are a number of sample runs of the S*A simulator system. Four short examples have been chosen to best represent the features of both the language and the simulator. Included with each example are some explanatory notes, an S*A description and a copy of the terminal session during simulation. All words enclosed in /* ... */ are comments and are ignored by the compiler and the simulator. These are included only for the convenience of the reader. In the terminal session listing, "Unix:" is the prompt by the operating system and "s->" is the prompt by the simulator. Expressions which were typed in by the user are *italicized*. Also included in the examples are sample requests for the statistics that are gathered.

1. Example 1. **If** and **while** statements

   This example is to demonstrate the use of **If** and **While** statements. It calculates the sum of the ones in the binary representation of a variable. This might be used, for example, in a parity calculation. The initial value of the variable "data" is set by the user via the **Write** command. During the calculation, the bit positions containing ones are printed out. After the "sum" has been calculated, it may be examined by the user via the **Read** command.

- S*A description -

```
sys calcsystem;  /* the outermost system */
 mech summer;  /* the summing mechanism */
    /* variable declarations */
    privar data    : seq (30 .. 0) bit;
    privar counter : seq (10 .. 0) bit;
    privar sum     : seq (10 .. 0) bit;
    privar bitsize : seq (10 .. 0) bit;
    proc looper();
       /* initialize counter and sum */
       sum := 0;
       counter := 0;
       bitsize := 30;
       break(1);    /* go into command mode */
                    /* to assign data       */
       /* loop through bit positions   */
       while (counter <= bitsize) do
          if (data and 1 ) =>
             /* rightmostbit is one */
             sum := sum + 1;
             print(counter);
          fi;
          data := shr data;
          counter := counter+1;
       od;
       break(2)   /* go into command mode to */
    endproc       /* examine sum             */
 endmech;
init summer.looper() /* start proc running */
endsys;
```

- Terminal Transcript -

```
Unix: parse <sim.ex.1 /* sim.ex.1 is S*A desc. */
  - mainline parser has started -
  parser completed successfully - rc=0

Unix:sim  /* run simulator */

s-> go;
  Break #1

s-> read /all value;
  data    : D/0  (31 bits)   /* "data" is 0 decimal */
  sum     : D/0  (11 bits)
  counter : D/0  (11 bits)
  bitsize : D/30 (11 bits)

s-> write data 131;
  data    : D/131 (31 bits)  /* "data" is 131 decimal */

s-> read data bin value ;
  data    : B/10000011 (31 bits) /* bin. rep of data */

s-> go;      /* run loop */
  Print counter = D/0  /* bit pos that contain ones */
  Print counter = D/1
  Print counter = D/7
  Break #2

s-> read sum value ;
  sum : D/3 (11 bits)   /* 3 ones in "data" */

s-> read sum stats ;  /* request stats on sum */
  sum: ,#R=3 ,#W=4  /* sum was read 3 times */
                    /* and written to 4 times */
s-> quit;
  --- sim is quitting --
```

2. Example 2. Passing of Parameters

This example demonstrates the passing of parameters, both input and output, via the subroutine **call** mechanism. In the example "iparm" is an input parameter, "oparm" is an output parameter and "ioparm" is both an input and output parameter. The user initially sets "iparm" and "ioparm" via the **write** command and after the simulation examines "oparm" and ioparm" via the **read** command.

- S*A description -

```
sys calcsystem;  /* the outermost system */
 mech parmtest;  /* the testing mechanism */
    /* variable declarations */
    privar iparm   : seq (30 .. 0) bit;
    privar oparm   : seq (30 .. 0) bit;
    privar ioparm  : seq (30 .. 0) bit;
    proc run();
       break(1);     /* go into command mode  */
                     /* to assign input parms */
       /* call subroutine  */
       call switcher.subr(<iparm,>oparm,|ioparm);
       break(2)      /* go into command mode  */
                     /* to examine output parms */
    endproc
 endmech
 mech switcher;      /* mech that switches parms */
    privar input  : seq (30 .. 0) bit;
    privar output : seq (30 .. 0) bit;
    privar ioput  : seq (30 .. 0) bit;
    privar temp   : seq (30 .. 0) bit;
    proc subr(<input,>output,|ioput) ;
        output := ioput;
        ioput := input;
        return
    endproc
 endmech;
init parmtest.run() /* start proc running */
endsys;
```

- Terminal Transcript -

```
Unix: parse <sim.ex.2 /* sim.ex.2 is S*A desc. */
  - mainline parser has started -
  parser completed successfully - rc=0

Unix:sim  /* run simulator */

s-> go;
  Break #1

s-> write iparm 10; write ioparm 20; write oparm 30;
  iparm   : D/10 (31 bits)   /* value is 10 decimal */
  ioparm  : D/20 (11 bits)
  oparm   : D/30 (11 bits)

s-> go;      /* run subroutine call */
  Break #2

s-> read iparm value;
  iparm :  D/10 (31 bits) /* values have been shifted */

s-> read ioparm value;
  ioparm :  D/10 (31 bits)

s-> read oparm value;
  oparm :  D/20 (31 bits)

s-> mech /all; /* request stats on mechanisms */
  mech-switcher: not active, #act= 1, #pending= 0
  mech-parmtest: active    , #act= 1, #pending= 0
   /* we see both mechs have been activated once */

s-> quit;
  --- sim is quitting --
```

3. Example 3. Contention for a Procedure

This example is to demonstrate how the S*A simulator handles the problem when a procedure is being simultaneously called from a number of points in the S*A description. The requests are queued up in the order that they are received and handled one at a time.

- S*A description -

```
sys calcsystem;  /* the outermost system */
 mech calltest;  /* the calling mechanism */
    /* variable declarations */
    privar one   : seq (10 .. 1) bit;
    privar two   : seq (10 .. 1) bit;
    privar three : seq (10 .. 1) bit;
    proc run();
       break(1);  /* put user into command mode  */
                  /* before parallel calls       */
       (one := 1; call contention.subr(<one) )
       box     /* parallel statement separator */
       (two := 2; call contention.subr(<two) )
       box
       (three := 3; call contention.subr(<three) )
       ; return
    endproc
 endmech
 mech contention ;    /* mechanism under contention */
    privar callnum : seq (10 .. 0) bit;
    proc subr(<callnum);
       print(callnum);
       break;
       return
    endproc
 endmech
 ; init calltest.run()   /* start proc running */
endsys;
```

- Terminal Transcript -

```
Unix: parse <sim.ex.3 /* sim.ex.3 is S*A desc. */
  - mainline parser has started -
  parser completed successfully - rc=0

Unix:sim  /* run simulator */

s-> go;
  Break #1

s-> go;      /* run simulation */
  print callnum = D/3  /* executed in arbitrary order */
  Break

s-> go;
  print callnum = D/2
  Break

  /* request info on mechanism "contention" */

s-> mech contention ;
  mech-contention: active, #act= 1, #pending= 2
  /* mech "contention" has been activated once and */
  /* has two pending calls to procdures within it   */

s-> go;
  print callnum = D/1
  Break

s-> go;

s-> quit;
  --- sim is quitting --
```

4. Example 4. **Sig** and **Await** statements

This example is to demonstrate the use of **sig** and **await** statements. There are two parallel control paths executing simultaneously, as initiated by the **act** statement. The **sig** and **await** statements on the two semaphores "leftsem" and "rightsem" are used to synchronize the two paths so that the **print** statments occur in the desired order.

- S*A description

```
sys calcsystem;  /* the outermost system */
 mech leftpath;  /* mech of the left par path */
    /* semaphore declarations */
    sync leftsem  : bit;
    sync rightsem : bit;
    proc pathl();
       /* start up right par path */
       act rightpath.pathr();
       /* left par path */
       await leftsem;  /* wait to start */
       print(1);
       sig rightsem;  /* let rhs run do #2 */
       await leftsem; /* wait for rhs to finish #2 */
       print(3);
       sig rightsem;  /* let rhs run do #4 */
       await leftsem; /* wait for rhs to finish #4 */
       print(5)
    endproc
 endmech
 mech rightpath;  /* mech of right par path */
    /* semaphore declarations */
    sync leftsem  : bit;
    sync rightsem : bit;
    proc pathr();
       /* right parallel path */
       sig leftsem;    /* let lhs do #1 */
       await rightsem; /* wait for lhs to finish #1 */
       print(2);
       sig leftsem;    /* let lhs do #3 */
       await rightsem; /* wait for lhs to finish #3 */
       print(4);
       sig leftsem;    /* let lhs do #5 */
       exit
    endproc
 endmech
 ; init leftpath.pathl()
endsys;
```

- Terminal Transcript -

```
Unix: parse <sim.ex.4 /* sim.ex.4 is S*A desc. */
  - mainline parser has started -
  parser completed successfully - rc=0

Unix:sim  /* run simulator */

s-> go;
  print #1
  print #2
  print #3
  print #4
  print #5

s-> quit;
  --- sim is quitting --
```